ESP32

ESP-IDF Programming Guide

Release v5.0.6-336-gfb12a68017
Espressif Systems
Apr 22, 2024

Table of contents

Table of contents i

1 Get Started 3
- 1.1 Introduction . . . 3
- 1.2 What You Need . . . 3
  - 1.2.1 Hardware . . . 3
  - 1.2.2 Software . . . 89
- 1.3 Installation . . . 89
  - 1.3.1 IDE . . . 89
  - 1.3.2 Manual Installation . . . 89
- 1.4 Build Your First Project . . . 119
- 1.5 Uninstall ESP-IDF . . . 119

2 API Reference 121
- 2.1 API Conventions . . . 121
  - 2.1.1 Error handling . . . 121
  - 2.1.2 Configuration structures . . . 121
  - 2.1.3 Private APIs . . . 122
  - 2.1.4 Components in example projects . . . 122
  - 2.1.5 API Stability . . . 123
- 2.2 Application Protocols . . . 124
  - 2.2.1 ASIO port . . . 124
  - 2.2.2 ESP-Modbus . . . 124
  - 2.2.3 ESP-MQTT . . . 125
  - 2.2.4 ESP-TLS . . . 140
  - 2.2.5 ESP HTTP Client . . . 155
  - 2.2.6 ESP Local Control . . . 170
  - 2.2.7 ESP Serial Slave Link . . . 179
  - 2.2.8 ESP x509 Certificate Bundle . . . 196
  - 2.2.9 HTTP Server . . . 198
  - 2.2.10 HTTPS Server . . . 225
  - 2.2.11 ICMP Echo . . . 229
  - 2.2.12 mDNS Service . . . 234
  - 2.2.13 Mbed TLS . . . 234
  - 2.2.14 IP Network Layer . . . 236
- 2.3 Bluetooth API . . . 236
  - 2.3.1 BT COMMON . . . 236
  - 2.3.2 BT LE . . . 245
  - 2.3.3 CLASSIC BT . . . 379
  - 2.3.4 Controller && VHCI . . . 522
  - 2.3.5 ESP-BLE-MESH . . . 532
  - 2.3.6 NimBLE-based host APIs . . . 814
- 2.4 Error Codes Reference . . . 816
- 2.5 Networking APIs . . . 823
  - 2.5.1 Wi-Fi . . . 823
  - 2.5.2 Ethernet . . . 944
  - 2.5.3 Thread . . . 980

2.5.4 ESP-NETIF . . . 987
2.5.5 IP Network Layer . . . 1019
2.5.6 Application Layer . . . 1021
2.6 Peripherals API . . . 1021
2.6.1 Analog to Digital Converter (ADC) Oneshot Mode Driver . . . 1021
2.6.2 Analog to Digital Converter (ADC) Continuous Mode Driver . . . 1030
2.6.3 Analog to Digital Converter (ADC) Calibration Driver . . . 1038
2.6.4 Clock Tree . . . 1042
2.6.5 Digital To Analog Converter (DAC) . . . 1049
2.6.6 GPIO & RTC GPIO . . . 1052
2.6.7 General Purpose Timer (GPTimer) . . . 1072
2.6.8 Inter-Integrated Circuit (I2C) . . . 1083
2.6.9 Inter-IC Sound (I2S) . . . 1100
2.6.10 LCD . . . 1135
2.6.11 LED Control (LEDC) . . . 1152
2.6.12 Motor Control Pulse Width Modulator (MCPWM) . . . 1171
2.6.13 Pulse Counter (PCNT) . . . 1223
2.6.14 Remote Control Transceiver (RMT) . . . 1238
2.6.15 SD Pull-up Requirements . . . 1265
2.6.16 SDMMC Host Driver . . . 1268
2.6.17 SD SPI Host Driver . . . 1274
2.6.18 SDIO Card Slave Driver . . . 1279
2.6.19 Sigma-Delta Modulation (SDM) . . . 1289
2.6.20 SPI Master Driver . . . 1294
2.6.21 SPI Slave Driver . . . 1319
2.6.22 ESP32-WROOM-32SE (Secure Element) . . . 1326
2.6.23 Touch Sensor . . . 1327
2.6.24 Two-Wire Automotive Interface (TWAI) . . . 1344
2.6.25 Universal Asynchronous Receiver/Transmitter (UART) . . . 1362
2.7 Project Configuration . . . 1387
2.7.1 Introduction . . . 1387
2.7.2 Project Configuration Menu . . . 1387
2.7.3 Using sdkconfig.defaults . . . 1387
2.7.4 Kconfig Formatting Rules . . . 1388
2.7.5 Backward Compatibility of Kconfig Options . . . 1388
2.7.6 Configuration Options Reference . . . 1388
2.8 Provisioning API . . . 1675
2.8.1 Protocol Communication . . . 1675
2.8.2 Unified Provisioning . . . 1690
2.8.3 Wi-Fi Provisioning . . . 1695
2.9 Storage API . . . 1714
2.9.1 FAT Filesystem Support . . . 1714
2.9.2 Manufacturing Utility . . . 1722
2.9.3 Non-volatile Storage Library . . . 1726
2.9.4 NVS Partition Generator Utility . . . 1748
2.9.5 SD/SDIO/MMC Driver . . . 1753
2.9.6 SPI Flash API . . . 1767
2.9.7 SPIFFS Filesystem . . . 1803
2.9.8 Virtual filesystem component . . . 1807
2.9.9 Wear Levelling API . . . 1823
2.10 System API . . . 1826
2.10.1 App Image Format . . . 1826
2.10.2 Application Level Tracing . . . 1831
2.10.3 Call function with external stack . . . 1836
2.10.4 Chip Revision . . . 1838
2.10.5 Console . . . 1840
2.10.6 eFuse Manager . . . 1849
2.10.7 Error Codes and Helper Functions . . . 1869

2.10.8 ESP HTTPS OTA . . . 1872
2.10.9 Event Loop Library . . . 1879
2.10.10 FreeRTOS (Overview) . . . 1891
2.10.11 FreeRTOS (ESP-IDF) . . . 1893
2.10.12 FreeRTOS (Supplemental Features) . . . 2011
2.10.13 Heap Memory Allocation . . . 2030
2.10.14 Heap Memory Debugging . . . 2043
2.10.15 High Resolution Timer (ESP Timer) . . . 2054
2.10.16 Internal and Unstable APIs . . . 2061
2.10.17 Inter-Processor Call . . . 2062
2.10.18 Interrupt allocation . . . 2067
2.10.19 Logging library . . . 2074
2.10.20 Miscellaneous System APIs . . . 2081
2.10.21 Over The Air Updates (OTA) . . . 2096
2.10.22 Performance Monitor . . . 2107
2.10.23 Power Management . . . 2110
2.10.24 POSIX Threads Support . . . 2116
2.10.25 Random Number Generation . . . 2120
2.10.26 Sleep Modes . . . 2122
2.10.27 SoC Capabilities . . . 2134
2.10.28 System Time . . . 2143
2.10.29 The himem allocation API . . . 2149
2.10.30 ULP Coprocessor programming . . . 2152
2.10.31 Watchdogs . . . 2184

3 Hardware Reference 2191
3.1 Chip Series Comparison . . . 2191
3.1.1 Related Documents . . . 2194

4 API Guides 2195
4.1 Application Level Tracing library . . . 2195
4.1.1 Overview . . . 2195
4.1.2 Modes of Operation . . . 2195
4.1.3 Configuration Options and Dependencies . . . 2196
4.1.4 How to Use This Library . . . 2197
4.2 Application Startup Flow . . . 2205
4.2.1 First stage bootloader . . . 2206
4.2.2 Second stage bootloader . . . 2206
4.2.3 Application startup . . . 2207
4.3 BluFi . . . 2208
4.3.1 Overview . . . 2208
4.3.2 The BluFi Flow . . . 2209
4.3.3 The Flow Chart of BluFi . . . 2209
4.3.4 The Frame Formats Defined in BluFi . . . 2209
4.3.5 The Security Implementation of ESP32 . . . 2215
4.3.6 GATT Related Instructions . . . 2216
4.4 Bootloader . . . 2216
4.4.1 Bootloader compatibility . . . 2217
4.4.2 Log Level . . . 2217
4.4.3 Factory reset . . . 2218
4.4.4 Boot from Test Firmware . . . 2218
4.4.5 Rollback . . . 2219
4.4.6 Watchdog . . . 2219
4.4.7 Bootloader Size . . . 2219
4.4.8 Fast boot from Deep Sleep . . . 2219
4.4.9 Custom bootloader . . . 2219
4.5 Build System . . . 2220
4.5.1 Overview . . . 2220

4.5.2 Using the Build System . . . 2221
4.5.3 Example Project . . . 2223
4.5.4 Project CMakeLists File . . . 2223
4.5.5 Component CMakeLists Files . . . 2225
4.5.6 Component Configuration . . . 2227
4.5.7 Preprocessor Definitions . . . 2227
4.5.8 Component Requirements . . . 2227
4.5.9 Overriding Parts of the Project . . . 2231
4.5.10 Configuration-Only Components . . . 2232
4.5.11 Debugging CMake . . . 2232
4.5.12 Example Component CMakeLists . . . 2233
4.5.13 Custom sdkconfig defaults . . . 2237
4.5.14 Flash arguments . . . 2237
4.5.15 Building the Bootloader . . . 2238
4.5.16 Writing Pure CMake Components . . . 2238
4.5.17 Using Third-Party CMake Projects with Components . . . 2238
4.5.18 Using Prebuilt Libraries with Components . . . 2239
4.5.19 Using ESP-IDF in Custom CMake Projects . . . 2239
4.5.20 ESP-IDF CMake Build System API . . . 2240
4.5.21 File Globbing & Incremental Builds . . . 2244
4.5.22 Build System Metadata . . . 2245
4.5.23 Build System Internals . . . 2245
4.5.24 Migrating from ESP-IDF GNU Make System . . . 2247
4.6 Core Dump . . . 2248
4.6.1 Overview . . . 2248
4.6.2 Configurations . . . 2249
4.6.3 Save core dump to flash . . . 2249
4.6.4 Print core dump to UART . . . 2250
4.6.5 ROM Functions in Backtraces . . . 2250
4.6.6 Dumping variables on demand . . . 2250
4.6.7 Running `espcoredump.py` . . . 2251
4.7 Deep Sleep Wake Stubs . . . 2254
4.7.1 Rules for Wake Stubs . . . 2254
4.7.2 Implementing A Stub . . . 2254
4.7.3 Loading Code Into RTC Memory . . . 2254
4.7.4 Loading Data Into RTC Memory . . . 2255
4.7.5 CRC Check For Wake Stubs . . . 2255
4.7.6 Example . . . 2255
4.8 Error Handling . . . 2256
4.8.1 Overview . . . 2256
4.8.2 Error codes . . . 2256
4.8.3 Converting error codes to error messages . . . 2256
4.8.4 `ESP_ERROR_CHECK` macro . . . 2257
4.8.5 `ESP_ERROR_CHECK_WITHOUT_ABORT` macro . . . 2257
4.8.6 `ESP_RETURN_ON_ERROR` macro . . . 2257
4.8.7 `ESP_GOTO_ON_ERROR` macro . . . 2257
4.8.8 `ESP_RETURN_ON_FALSE` macro . . . 2257
4.8.9 `ESP_GOTO_ON_FALSE` macro . . . 2257
4.8.10 `CHECK MACROS Examples` . . . 2258
4.8.11 Error handling patterns . . . 2258
4.8.12 C++ Exceptions . . . 2259
4.9 ESP-BLE-MESH . . . 2259
4.9.1 Getting Started with ESP-BLE-MESH . . . 2260
4.9.2 ESP-BLE-MESH Examples . . . 2266
4.9.3 ESP-BLE-MESH Demo Videos . . . 2267
4.9.4 ESP-BLE-MESH FAQ . . . 2267
4.9.5 Related Documents . . . 2267
4.10 ESP-WIFI-MESH . . . 2296

4.10.1 Overview . . . 2296
4.10.2 Introduction . . . 2296
4.10.3 ESP-WIFI-MESH Concepts . . . 2297
4.10.4 Building a Network . . . 2303
4.10.5 Managing a Network . . . 2308
4.10.6 Data Transmission . . . 2311
4.10.7 Channel Switching . . . 2313
4.10.8 Performance . . . 2316
4.10.9 Further Notes . . . 2317
4.11 Event Handling . . . 2317
4.11.1 Wi-Fi, Ethernet, and IP Events . . . 2317
4.11.2 Mesh Events . . . 2318
4.11.3 Bluetooth Events . . . 2319
4.12 Fatal Errors . . . 2319
4.12.1 Overview . . . 2319
4.12.2 Panic Handler . . . 2319
4.12.3 Register Dump and Backtrace . . . 2320
4.12.4 GDB Stub . . . 2322
4.12.5 RTC Watchdog Timeout . . . 2323
4.12.6 Guru Meditation Errors . . . 2323
4.12.7 Other Fatal Errors . . . 2325
4.13 Flash Encryption . . . 2327
4.13.1 Introduction . . . 2327
4.13.2 Relevant eFuses . . . 2328
4.13.3 Flash Encryption Process . . . 2328
4.13.4 Flash Encryption Configuration . . . 2329
4.13.5 Possible Failures . . . 2335
4.13.6 ESP32 Flash Encryption Status . . . 2336
4.13.7 Reading and Writing Data in Encrypted Flash . . . 2337
4.13.8 Updating Encrypted Flash . . . 2338
4.13.9 Disabling Flash Encryption . . . 2338
4.13.10 Key Points About Flash Encryption . . . 2338
4.13.11 Limitations of Flash Encryption . . . 2339
4.13.12 Flash Encryption and Secure Boot . . . 2339
4.13.13 Advanced Features . . . 2339
4.13.14 Technical Details . . . 2341
4.14 Hardware Abstraction . . . 2342
4.14.1 Architecture . . . 2342
4.14.2 LL (Low Level) Layer . . . 2343
4.14.3 HAL (Hardware Abstraction Layer) . . . 2344
4.15 High-Level Interrupts . . . 2345
4.15.1 Interrupt Levels . . . 2345
4.15.2 Notes . . . 2346
4.16 JTAG Debugging . . . 2346
4.16.1 Introduction . . . 2347
4.16.2 How it Works? . . . 2347
4.16.3 Selecting JTAG Adapter . . . 2348
4.16.4 Setup of OpenOCD . . . 2348
4.16.5 Configuring ESP32 Target . . . 2349
4.16.6 Launching Debugger . . . 2354
4.16.7 Debugging Examples . . . 2354
4.16.8 Building OpenOCD from Sources . . . 2354
4.16.9 Tips and Quirks . . . 2359
4.16.10 Related Documents . . . 2364
4.17 Linker Script Generation . . . 2389
4.17.1 Overview . . . 2389
4.17.2 Quick Start . . . 2389
4.17.3 Linker Script Generation Internals . . . 2392

4.18 lwIP . . . 2399
4.18.1 Supported APIs . . . 2399
4.18.2 BSD Sockets API . . . 2399
4.18.3 Netconn API . . . 2403
4.18.4 lwIP FreeRTOS Task . . . 2403
4.18.5 IPv6 Support . . . 2404
4.18.6 esp-lwip custom modifications . . . 2404
4.18.7 Performance Optimization . . . 2406
4.19 Memory Types . . . 2407
4.19.1 DRAM (Data RAM) . . . 2407
4.19.2 IRAM (Instruction RAM) . . . 2408
4.19.3 IROM (code executed from flash) . . . 2409
4.19.4 DROM (data stored in flash) . . . 2409
4.19.5 RTC Slow memory . . . 2409
4.19.6 RTC FAST memory . . . 2410
4.19.7 DMA Capable Requirement . . . 2410
4.19.8 DMA Buffer in the stack . . . 2410
4.20 OpenThread . . . 2411
4.20.1 Modes of the OpenThread stack . . . 2411
4.20.2 How to Write an OpenThread Application . . . 2411
4.20.3 The OpenThread Border Router . . . 2413
4.21 Partition Tables . . . 2413
4.21.1 Overview . . . 2413
4.21.2 Built-in Partition Tables . . . 2413
4.21.3 Creating Custom Tables . . . 2414
4.21.4 Generating Binary Partition Table . . . 2416
4.21.5 Partition Size Checks . . . 2416
4.21.6 Flashing the partition table . . . 2417
4.21.7 Partition Tool (parttool.py) . . . 2417
4.22 Performance . . . 2419
4.22.1 How to Optimize Performance . . . 2419
4.22.2 Guides . . . 2419
4.23 RF calibration . . . 2437
4.23.1 Partial calibration . . . 2437
4.23.2 Full calibration . . . 2437
4.23.3 No calibration . . . 2437
4.23.4 PHY initialization data . . . 2437
4.23.5 API Reference . . . 2438
4.24 Secure Boot . . . 2440
4.24.1 Background . . . 2441
4.24.2 Secure Boot Process Overview . . . 2441
4.24.3 Keys . . . 2441
4.24.4 Bootloader Size . . . 2442
4.24.5 How To Enable Secure Boot . . . 2442
4.24.6 Re-Flashable Software Bootloader . . . 2443
4.24.7 Generating Secure Boot Signing Key . . . 2443
4.24.8 Remote Signing of Images . . . 2444
4.24.9 Secure Boot Best Practices . . . 2444
4.24.10 Technical Details . . . 2444
4.24.11 Secure Boot & Flash Encryption . . . 2446
4.24.12 Signed App Verification Without Hardware Secure Boot . . . 2446
4.24.13 Advanced Features . . . 2446
4.25 Secure Boot V2 . . . 2447
4.25.1 Background . . . 2447
4.25.2 Advantages . . . 2447
4.25.3 Secure Boot V2 Process . . . 2447
4.25.4 Signature Block Format . . . 2448
4.25.5 Secure Padding . . . 2448

4.25.6 Verifying a Signature Block . . . 2449
4.25.7 Verifying an Image . . . 2449
4.25.8 Bootloader Size . . . 2449
4.25.9 eFuse usage . . . 2449
4.25.10 How To Enable Secure Boot V2 . . . 2450
4.25.11 Restrictions after Secure Boot is enabled . . . 2450
4.25.12 Generating Secure Boot Signing Key . . . 2451
4.25.13 Remote Signing of Images . . . 2451
4.25.14 Secure Boot Best Practices . . . 2452
4.25.15 Technical Details . . . 2452
4.25.16 Secure Boot & Flash Encryption . . . 2452
4.25.17 Signed App Verification Without Hardware Secure Boot . . . 2452
4.25.18 Advanced Features . . . 2453
4.26 Support for External RAM . . . 2453
4.26.1 Introduction . . . 2453
4.26.2 Hardware . . . 2454
4.26.3 Configuring External RAM . . . 2454
4.26.4 Restrictions . . . 2455
4.26.5 Failure to initialize . . . 2456
4.26.6 Chip Revisions . . . 2456
4.27 Thread Local Storage . . . 2457
4.27.1 Overview . . . 2457
4.27.2 FreeRTOS Native API . . . 2457
4.27.3 Pthread API . . . 2457
4.27.4 C11 Standard . . . 2457
4.28 Tools . . . 2458
4.28.1 IDF Frontend - idf.py . . . 2458
4.28.2 IDF Docker Image . . . 2462
4.28.3 IDF Windows Installer . . . 2464
4.28.4 IDF Component Manager . . . 2465
4.28.5 IDF Clang Tidy . . . 2466
4.28.6 Downloadable Tools . . . 2467
4.29 Unit Testing in ESP32 . . . 2480
4.29.1 Normal Test Cases . . . 2480
4.29.2 Multi-device Test Cases . . . 2481
4.29.3 Multi-stage Test Cases . . . 2482
4.29.4 Tests For Different Targets . . . 2482
4.29.5 Building Unit Test App . . . 2483
4.29.6 Running Unit Tests . . . 2483
4.29.7 Timing Code with Cache Compensated Timer . . . 2484
4.29.8 Mocks . . . 2485
4.30 Unit Testing on Linux . . . 2487
4.30.1 Embedded Software Tests . . . 2487
4.30.2 IDF Unit Tests on Linux Host . . . 2488
4.31 Wi-Fi Driver . . . 2489
4.31.1 ESP32 Wi-Fi Feature List . . . 2489
4.31.2 How To Write a Wi-Fi Application . . . 2489
4.31.3 ESP32 Wi-Fi API Error Code . . . 2490
4.31.4 ESP32 Wi-Fi API Parameter Initialization . . . 2490
4.31.5 ESP32 Wi-Fi Programming Model . . . 2490
4.31.6 ESP32 Wi-Fi Event Description . . . 2491
4.31.7 ESP32 Wi-Fi Station General Scenario . . . 2494
4.31.8 ESP32 Wi-Fi AP General Scenario . . . 2497
4.31.9 ESP32 Wi-Fi Scan . . . 2497
4.31.10 ESP32 Wi-Fi Station Connecting Scenario . . . 2504
4.31.11 ESP32 Wi-Fi Station Connecting When Multiple APs Are Found . . . 2511
4.31.12 Wi-Fi Reconnect . . . 2511
4.31.13 Wi-Fi Beacon Timeout . . . 2511

4.31.14 ESP32 Wi-Fi Configuration . . . 2511
4.31.15 Wi-Fi Easy Connect™ (DPP) . . . 2517
4.31.16 Wireless Network Management . . . 2518
4.31.17 Radio Resource Measurement . . . 2518
4.31.18 Fast BSS Transition . . . 2519
4.31.19 ESP32 Wi-Fi Power-saving Mode . . . 2519
4.31.20 ESP32 Wi-Fi Throughput . . . 2521
4.31.21 Wi-Fi 80211 Packet Send . . . 2521
4.31.22 Wi-Fi Sniffer Mode . . . 2522
4.31.23 Wi-Fi Multiple Antennas . . . 2523
4.31.24 Wi-Fi Channel State Information . . . 2524
4.31.25 Wi-Fi Channel State Information Configure . . . 2526
4.31.26 Wi-Fi HT20/40 . . . 2526
4.31.27 Wi-Fi QoS . . . 2526
4.31.28 Wi-Fi AMSDU . . . 2527
4.31.29 Wi-Fi Fragment . . . 2527
4.31.30 WPS Enrollee . . . 2527
4.31.31 Wi-Fi Buffer Usage . . . 2527
4.31.32 How to Improve Wi-Fi Performance . . . 2528
4.31.33 Wi-Fi Menuconfig . . . 2531
4.31.34 Troubleshooting . . . 2534
4.32 Wi-Fi Security . . . 2537
4.32.1 ESP32 Wi-Fi Security Features . . . 2537
4.32.2 Protected Management Frames (PMF) . . . 2540
4.32.3 WiFi Enterprise . . . 2541
4.32.4 WPA3-Personal . . . 2541
4.32.5 Wi-Fi Enhanced Open™ . . . 2542
4.33 RF Coexistence . . . 2542
4.33.1 Overview . . . 2543
4.33.2 Supported Coexistence Scenario for ESP32 . . . 2543
4.33.3 Coexistence Mechanism and Policy . . . 2543
4.33.4 How to Use the Coexistence Feature . . . 2545
4.34 Reproducible Builds . . . 2546
4.34.1 Introduction . . . 2546
4.34.2 Reasons for non-reproducible builds . . . 2546
4.34.3 Enabling reproducible builds in ESP-IDF . . . 2546
4.34.4 How reproducible builds are achieved . . . 2546
4.34.5 Reproducible builds and debugging . . . 2547
4.34.6 Factors which still affect reproducible builds . . . 2547
4.35 Low Power Mode User Guide . . . 2547

5 Migration Guides 2549
5.1 ESP-IDF 5.x Migration Guide . . . 2549
5.1.1 Migration from 4.4 to 5.0 . . . 2549

6 Libraries and Frameworks 2581
6.1 Cloud Frameworks . . . 2581
6.1.1 ESP RainMaker . . . 2581
6.1.2 AWS IoT . . . 2581
6.1.3 Azure IoT . . . 2581
6.1.4 Google IoT Core . . . 2581
6.1.5 Aliyun IoT . . . 2581
6.1.6 Joylink IoT . . . 2581
6.1.7 Tencent IoT . . . 2582
6.1.8 Tencentyun IoT . . . 2582
6.1.9 Baidu IoT . . . 2582
6.2 Espressif ' s Frameworks . . . 2582
6.2.1 Espressif Audio Development Framework . . . 2582

6.2.2 ESP-CSI . . . 2582
6.2.3 Espressif DSP Library . . . 2582
6.2.4 ESP-WIFI-MESH Development Framework . . . 2583
6.2.5 ESP-WHO . . . 2583
6.2.6 ESP RainMaker . . . 2583
6.2.7 ESP-IoT-Solution . . . 2583
6.2.8 ESP-Protocols . . . 2583
6.2.9 ESP-BSP . . . 2584

7 Contributions Guide **2585**
7.1 How to Contribute . . . 2585
7.2 Before Contributing . . . 2585
7.3 Pull Request Process . . . 2585
7.4 Legal Part . . . 2586
7.5 Related Documents . . . 2586
7.5.1 Espressif IoT Development Framework Style Guide . . . 2586
7.5.2 Install pre-commit Hook for ESP-IDF Project . . . 2593
7.5.3 Documenting Code . . . 2594
7.5.4 Creating Examples . . . 2599
7.5.5 API Documentation Template . . . 2600
7.5.6 Contributor Agreement . . . 2602
7.5.7 Copyright Header Guide . . . 2604
7.5.8 ESP-IDF Tests with Pytest Guide . . . 2605

8 ESP-IDF Versions **2615**
8.1 Releases . . . 2615
8.2 Which Version Should I Start With? . . . 2615
8.3 Versioning Scheme . . . 2615
8.4 Support Periods . . . 2616
8.5 Checking the Current Version . . . 2617
8.6 Git Workflow . . . 2618
8.7 Updating ESP-IDF . . . 2618
8.7.1 Updating to Stable Release . . . 2619
8.7.2 Updating to a Pre-Release Version . . . 2619
8.7.3 Updating to Master Branch . . . 2619
8.7.4 Updating to a Release Branch . . . 2620

9 Resources **2621**
9.1 PlatformIO . . . 2621
9.1.1 What is PlatformIO? . . . 2621
9.1.2 Installation . . . 2621
9.1.3 Configuration . . . 2622
9.1.4 Tutorials . . . 2622
9.1.5 Project Examples . . . 2622
9.1.6 Next Steps . . . 2622
9.2 Useful Links . . . 2622

10 Copyrights and Licenses **2623**
10.1 Software Copyrights . . . 2623
10.1.1 Firmware Components . . . 2623
10.1.2 Documentation . . . 2624
10.2 ROM Source Code Copyrights . . . 2624
10.3 Xtensa libhal MIT License . . . 2625
10.4 TinyBasic Plus MIT License . . . 2625
10.5 TJpgDec License . . . 2625

11 About **2627**

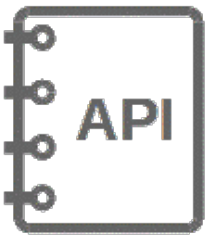
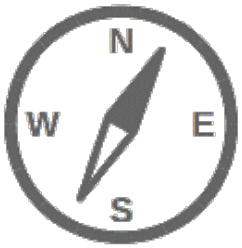
12 Switch Between Languages **2629**

Index **2631**

Index **2631**

This is the documentation for Espressif IoT Development Framework (esp-idf). ESP-IDF is the official development framework for the ESP32, ESP32-S and ESP32-C Series SoCs.

This document describes using ESP-IDF with the ESP32 SoC.

		
Get Started	API Reference	API Guides

Chapter 1

Get Started

This document is intended to help you set up the software development environment for the hardware based on the ESP32 chip by Espressif. After that, a simple example will show you how to use ESP-IDF (Espressif IoT Development Framework) for menu configuration, then for building and flashing firmware onto an ESP32 board.

Note: This is documentation for branch `release/v5.0` of ESP-IDF. Other *ESP-IDF Versions* are also available.

1.1 Introduction

ESP32 is a system on a chip that integrates the following features:

- Wi-Fi (2.4 GHz band)
- Bluetooth
- Dual high performance Xtensa® 32-bit LX6 CPU cores
- Ultra Low Power co-processor
- Multiple peripherals

Powered by 40 nm technology, ESP32 provides a robust, highly integrated platform, which helps meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.

Espressif provides basic hardware and software resources to help application developers realize their ideas using the ESP32 series hardware. The software development framework by Espressif is intended for development of Internet-of-Things (IoT) applications with Wi-Fi, Bluetooth, power management and several other system features.

1.2 What You Need

1.2.1 Hardware

- An **ESP32** board.
- **USB cable** - USB A / micro USB B.
- **Computer** running Windows, Linux, or macOS.

Note: Currently, some of the development boards are using USB Type C connectors. Be sure you have the correct cable to connect your board!

If you have one of ESP32 official development boards listed below, you can click on the link to learn more about the hardware.

ESP32-DevKitC V4 Getting Started Guide

This guide shows how to start using the ESP32-DevKitC V4 development board.

What You Need

- *ESP32-DevKitC V4 board*
- USB A / micro USB B cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-DevKitC V4 is a small-sized ESP32-based development board produced by Espressif. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-DevKitC V4 on a breadboard.

To cover a wide range of user requirements, the following versions of ESP32-DevKitC V4 are available:

- different ESP32 modules
  - ESP32-WROOM-DA
  - ESP32-WROOM-32E
  - ESP32-WROOM-32UE
  - ESP32-WROOM-32D
  - ESP32-WROOM-32U
  - ESP32-SOLO-1
  - ESP32-WROVER-E
  - ESP32-WROVER-IE
- male or female pin headers.

For details please refer to ESP Product Selector.

Functional Description The following figure and the table below describe the key components, interfaces and controls of the ESP32-DevKitC V4 board.

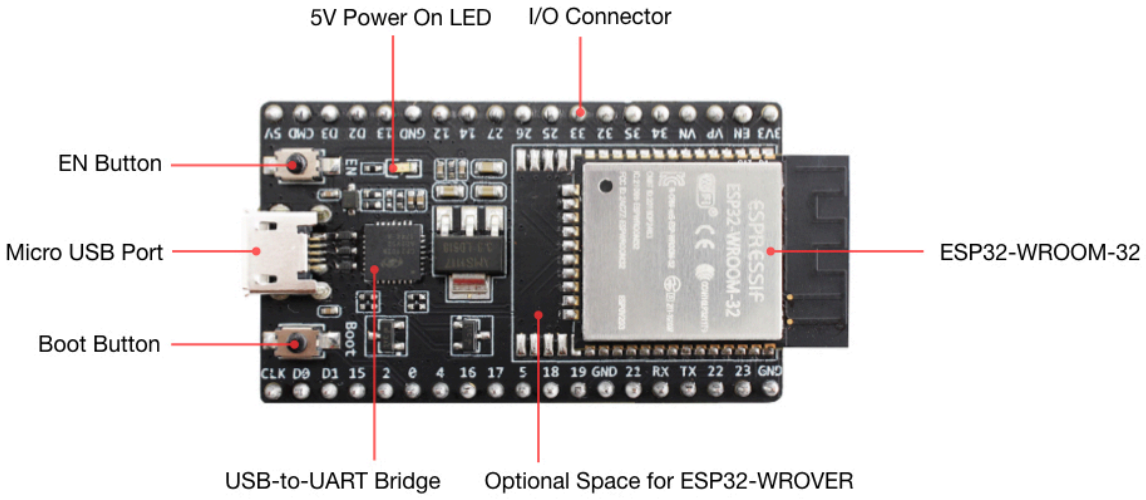


Fig. 1: ESP32-DevKitC V4 with ESP32-WROOM-32 module soldered

Key Component	Description
ESP32-WROOM-32	A module with ESP32 at its core. For more information, see ESP32-WROOM-32 Datasheet.
EN	Reset button.
Boot	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
USB-to-UART Bridge	Single USB-UART bridge chip provides transfer rates of up to 3 Mbps.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32-WROOM-32 module.
5V Power On LED	Turns on when the USB or an external 5V power supply is connected to the board. For details see the schematics in *Related Documents*.
I/O	Most of the pins on the ESP module are broken out to the pin headers on the board. You can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V / GND header pins
- 3V3 / GND header pins

Warning: The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.

Header Block The two tables below provide the **Name** and **Function** of I/O header pins on both sides of the board, as shown in *ESP32-DevKitC V4 with ESP32-WROOM-32 module soldered*.

J2

No.	Name	Type[1]	Function
1	3V3	P	3.3 V power supply
2	EN	I	CHIP_PU, Reset
3	VP	I	GPIO36, ADC1_CH0, S_VP
4	VN	I	GPIO39, ADC1_CH3, S_VN
5	IO34	I	GPIO34, ADC1_CH6, VDET_1
6	IO35	I	GPIO35, ADC1_CH7, VDET_2
7	IO32	I/O	GPIO32, ADC1_CH4, TOUCH_CH9, XTAL_32K_P
8	IO33	I/O	GPIO33, ADC1_CH5, TOUCH_CH8, XTAL_32K_N
9	IO25	I/O	GPIO25, ADC1_CH8, DAC_1
10	IO26	I/O	GPIO26, ADC2_CH9, DAC_2
11	IO27	I/O	GPIO27, ADC2_CH7, TOUCH_CH7
12	IO14	I/O	GPIO14, ADC2_CH6, TOUCH_CH6, MTMS
13	IO12	I/O	GPIO12, ADC2_CH5, TOUCH_CH5, MTDI
14	GND	G	Ground
15	IO13	I/O	GPIO13, ADC2_CH4, TOUCH_CH4, MTCK
16	D2	I/O	GPIO9, D2[2]
17	D3	I/O	GPIO10, D3[2]
18	CMD	I/O	GPIO11, CMD[2]
19	5V	P	5 V power supply

[1] P: Power supply; I: Input; O: Output.

[2] The pins D0, D1, D2, D3, CMD and CLK are used internally for communication between ESP32 and SPI flash memory. They are grouped on both sides near the USB connector. Avoid using these pins, as it may disrupt access to the SPI flash memory / SPI RAM.

J3

No.	Name	Type[?]	Function
1	GND	G	Ground
2	IO23	I/O	GPIO23
3	IO22	I/O	GPIO22
4	TX	I/O	GPIO1, U0TXD
5	RX	I/O	GPIO3, U0RXD
6	IO21	I/O	GPIO21
7	GND	G	Ground
8	IO19	I/O	GPIO19
9	IO18	I/O	GPIO18
10	IO5	I/O	GPIO5
11	IO17	I/O	GPIO17[3]
12	IO16	I/O	GPIO16[Page 6, 3]
13	IO4	I/O	GPIO4, ADC2_CH0, TOUCH_CH0
14	IO0	I/O	GPIO0, ADC2_CH1, TOUCH_CH1, Boot
16	IO2	I/O	GPIO2, ADC2_CH2, TOUCH_CH2
17	IO15	I/O	GPIO15, ADC2_CH3, TOUCH_CH3, MTDO
17	D1	I/O	GPIO8, D1[?]
18	D0	I/O	GPIO7, D0[?]
19	CLK	I/O	GPIO6, CLK[?]

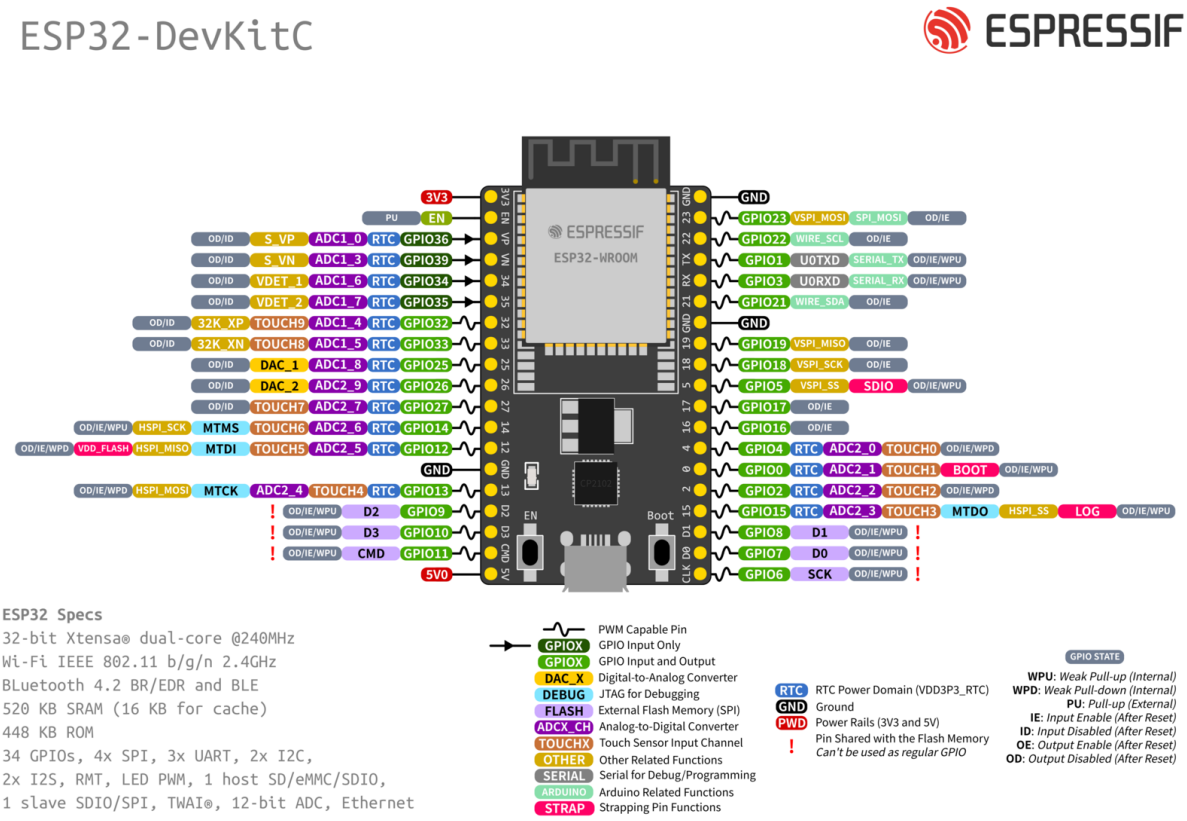


Fig. 2: ESP32-DevKitC Pin Layout (click to enlarge)

Pin Layout

Note on C15 The component C15 may cause the following issues on earlier ESP32-DevKitC V4 boards:

[3] The pins GPIO16 and GPIO17 are available for use only on the boards with the modules ESP32-WROOM and ESP32-SOLO-1. The boards with ESP32-WROVER modules have the pins reserved for internal use.

- The board may boot into Download mode
- If you output clock on GPIO0, C15 may impact the signal

In case these issues occur, please remove the component. The figure below shows the location of C15 highlighted in yellow.

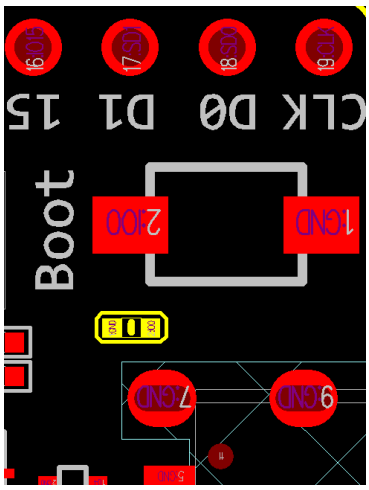


Fig. 3: Location of C15 (yellow) on ESP32-DevKitC V4 board

Start Application Development Before powering up your ESP32-DevKitC V4, please make sure that the board is in good condition with no obvious signs of damage.

After that, proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

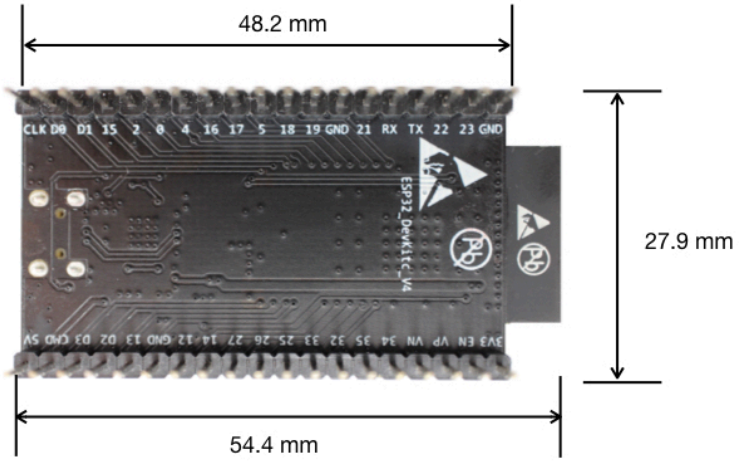


Fig. 4: Dimensions of ESP32-DevKitC board with ESP32-WROOM-32 module soldered - back (click to enlarge)

Board Dimensions

Related Documents

- ESP32-DevKitC V4 schematics (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROOM-32 Datasheet (PDF)
- ESP32-WROOM-32D and ESP32-WROOM-32U Datasheet (PDF)
- ESP32-WROOM-DA Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- ESP32-WROVER-B Datasheet (PDF)
- ESP Product Selector

For further design documentation for the board, please contact us at sales@espressif.com.

ESP32-DevKitC V2 Getting Started Guide

This guide shows how to start using the ESP32-DevKitC V2 development board.

What You Need

- *ESP32-DevKitC V2 board*
- USB A / micro USB B cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-DevKitC V2 is a small-sized ESP32-based development board produced by Espressif. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-DevKitC V4 on a breadboard.

Functional Description The following figure and the table below describe the key components, interfaces and controls of the ESP32-DevKitC V2 board.

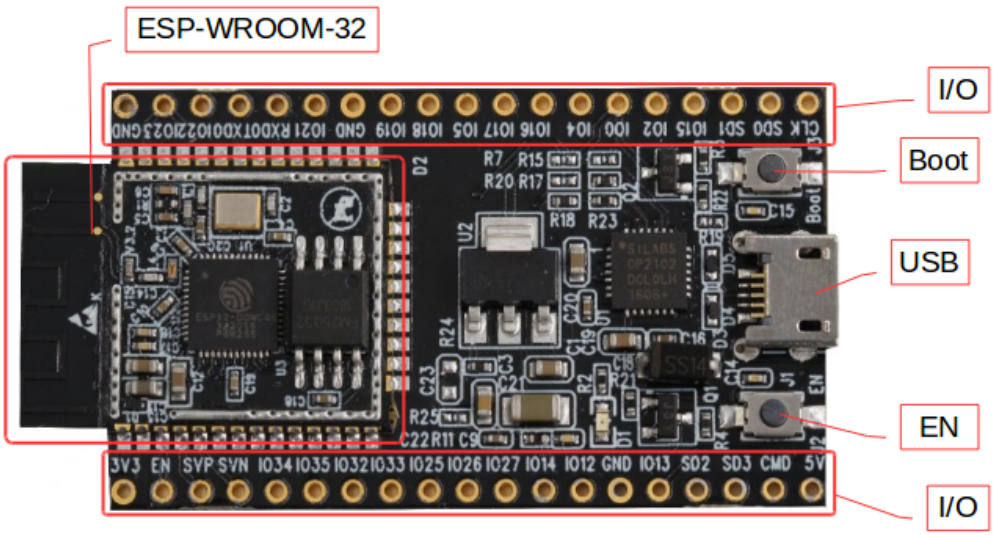


Fig. 5: ESP32-DevKitC V2 board layout

Key Component	Description
ESP32-WROOM-32	Standard module with ESP32 at its core. For more information, see ESP32-WROOM-32 Datasheet
EN	Reset button.
Boot	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and ESP32-WROOM-32.
I/O	Most of the pins on the ESP module are broken out to the pin headers on the board. You can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V / GND header pins
- 3V3 / GND header pins

Warning: The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.

Start Application Development Before powering up your ESP32-DevKitC V2, please make sure that the board is in good condition with no obvious signs of damage.

After that, proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- ESP32-DevKitC schematics (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROOM-32 Datasheet (PDF)

ESP-WROVER-KIT V4.1 Getting Started Guide

This guide shows how to get started with the ESP-WROVER-KIT V4.1 development board and also provides information about its functionality and configuration options.

What You Need

- *ESP-WROVER-KIT V4.1 board*
- USB 2.0 cable（A to Micro-B）
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP-WROVER-KIT is an ESP32-based development board produced by Espressif.

ESP-WROVER-KIT features the following integrated components:

- ESP32-WROVER-E module
- LCD screen
- microSD card slot

Its another distinguishing feature is the embedded FTDI FT2232HL chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface without a separate JTAG debugger. ESP-WROVER-KIT makes development convenient, easy, and cost-effective.

Most of the ESP32 I/O pins are broken out to the board's pin headers for easy access.

Note: ESP32's GPIO16 and GPIO17 are used as chip select and clock signals for PSRAM. By default, the two GPIOs are not broken out to the board's pin headers in order to ensure reliable performance.

Functionality Overview The block diagram below shows the main components of ESP-WROVER-KIT and their interconnections.

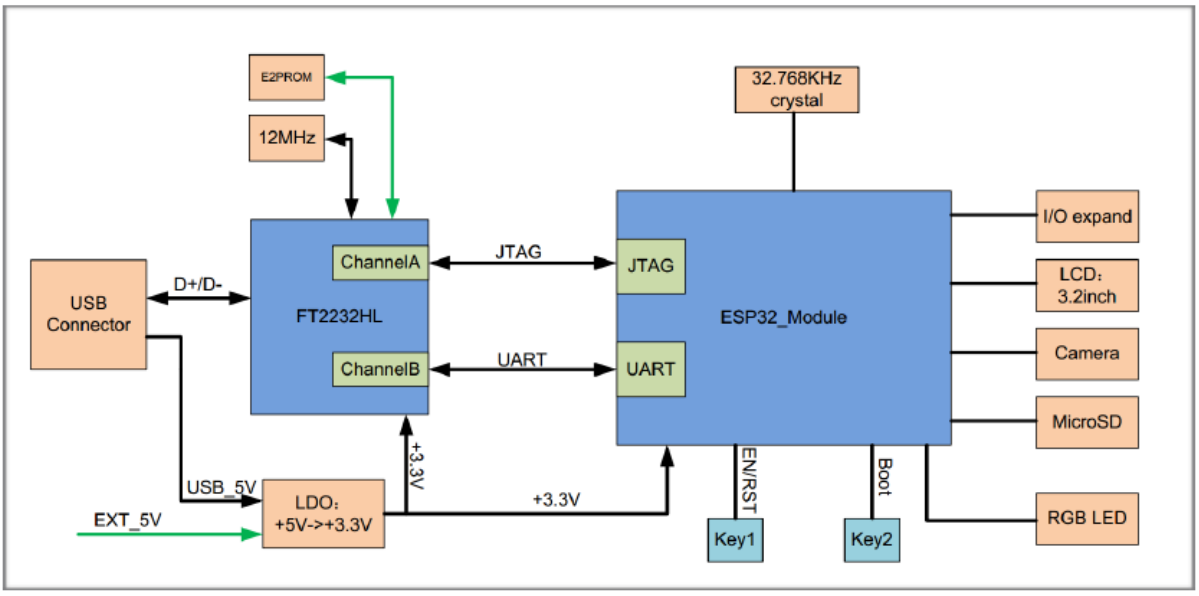


Fig. 6: ESP-WROVER-KIT block diagram

Functional Description The following two figures and the table below describe the key components, interfaces, and controls of the ESP-WROVER-KIT board.

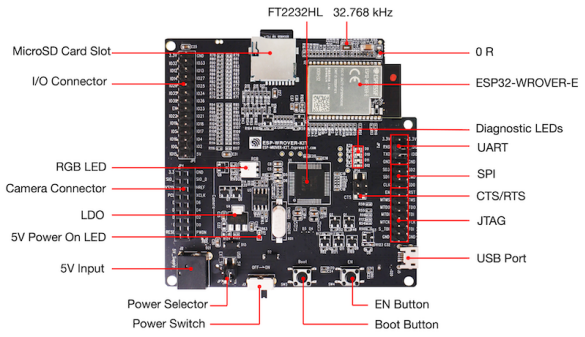


Fig. 7: ESP-WROVER-KIT board layout - front

The table below provides description in the following manner:

- Starting from the first picture's top right corner and going clockwise
- Then moving on to the second picture

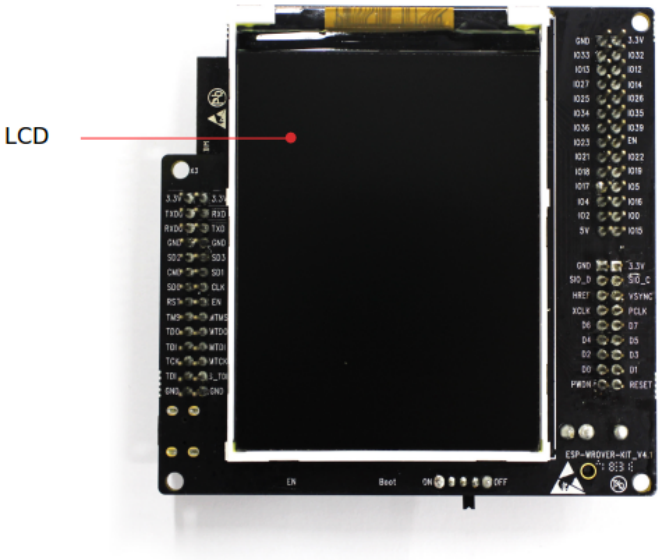
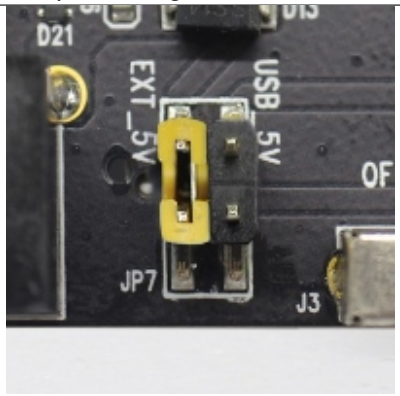
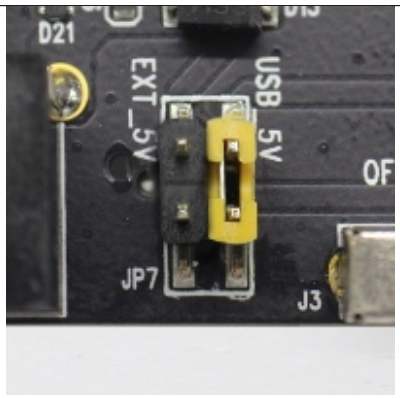
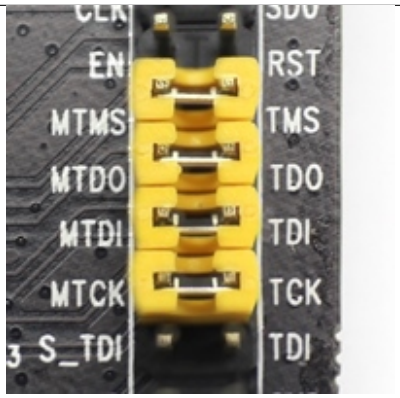
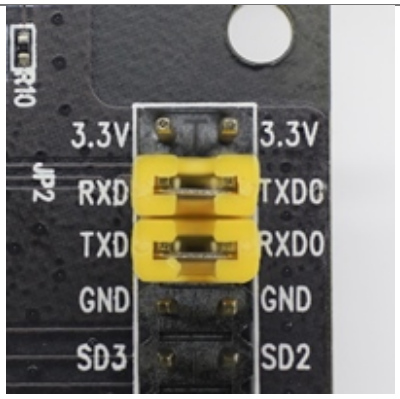
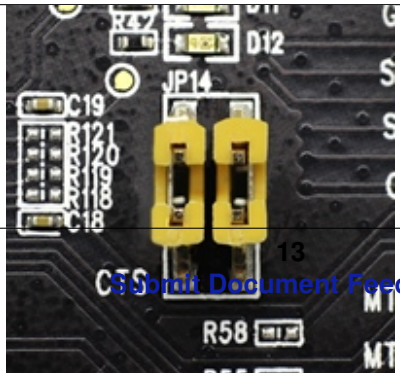


Fig. 8: ESP-WROVER-KIT board layout - back

Key Component	Description
FT2232HL	The FT2232HL chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232HL also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232HL chip enhances user-friendliness in terms of application development and debugging. See ESP-WROVER-KIT V4.1 schematic.
32.768 kHz	External precision 32.768 kHz crystal oscillator serves as a clock with low-power consumption while the chip is in Deep-sleep mode.
0R	Zero-ohm resistor intended as a placeholder for a current shunt, can be desoldered or replaced with a current shunt to facilitate the measurement of ESP32's current consumption in different modes.
ESP32-WROVER-E module	This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities.
Diagnostic LEDs	Four red LEDs connected to the GPIO pins of FT2232HL. Intended for future use.
UART	Serial port. The serial TX/RX signals of FT2232HL and ESP32 are broken out to the inward and outward sides of JP2 respectively. By default, these pairs of pins are connected with jumpers. To use ESP32's serial interface, remove the jumpers and connect another external serial device to the respective pins.
SPI	By default, ESP32 uses its SPI interface to access flash and PSRAM memory inside the module. Use these pins to connect ESP32 to another SPI device. In this case, an extra chip select (CS) signal is needed. Please note that the voltage of this interface is 3.3 V.
CTS/RTS	Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, short the respective pins of JP14 with jumpers.
JTAG	JTAG interface. JTAG signals of FT2232HL and ESP32 are broken out to the inward and outward sides of JP2 respectively. By default, these pairs of pins are disconnected. To enable JTAG, short the respective pins with jumpers as shown in Section *Setup Options*.
USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
EN Button	Reset button.
BOOT Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
Power Switch	Power On/Off Switch. Toggling toward the **Boot** button powers the board on, toggling away from **Boot** powers the board off.
Power Selector	Power supply selector interface. The board can be powered either via USB or via the 5V Input interface. Select the power source with a jumper. For more details, see Section *Setup Options*, jumper header JP7.

Setup Options There are three jumper blocks available to set up the board functionality. The most frequently required options are listed in the table below.

Header	Jumper Setting	Description of Functionality
JP7		Power ESP-WROVER-KIT via an external power supply
JP7		Power ESP-WROVER-KIT via USB
JP2		Enable JTAG functionality
JP2		Enable UART communication
		

Submit Document Feedback

Allocation of ESP32 Pins Some pins or terminals of ESP32 are allocated for use with the onboard or external hardware. If that hardware is not used, e.g., nothing is plugged into the Camera (JP4) header, then these GPIOs can be used for other purposes.

Some of the pins, such as GPIO0 or GPIO2, have multiple functions and some of them are shared among onboard and external peripheral devices. Certain combinations of peripherals cannot work together. For example, it is not possible to do JTAG debugging of an application that is using SD card, because several pins are shared by JTAG and the SD card slot.

In other cases, peripherals can coexist under certain conditions. This is applicable to, for example, LCD screen and SD card that share only a single pin GPIO21. This pin is used to provide D/C (Data/Control) signal for the LCD as well as the Card Detect signal read from the SD card slot. If the card detect functionality is not essential, then it may be disabled by removing R167, so both LCD and SD may operate together.

For more details on which pins are shared among which peripherals, please refer to the table in the next section.

Main I/O Connector / JP1 The JP1 connector consists of 14x2 male pins whose functions are shown in the middle two "I/O" columns of the table below. The two "Shared With" columns on both sides describe where else on the board a certain GPIO is used.

Shared With	I/O	I/O	Shared With
n/a	3.3V	GND	n/a
NC/XTAL	IO32	IO33	NC/XTAL
JTAG, microSD	IO12	IO13	JTAG, microSD
JTAG, microSD	IO14	IO27	Camera
Camera	IO26	IO25	Camera, LCD
Camera	IO35	IO34	Camera
Camera	IO39	IO36	Camera
JTAG	EN	IO23	Camera, LCD
Camera, LCD	IO22	IO21	Camera, LCD, microSD
Camera, LCD	IO19	IO18	Camera, LCD
Camera, LCD	IO5	IO17	PSRAM
PSRAM	IO16	IO4	LED, Camera, microSD
Camera, LED, Boot	IO0	IO2	LED, microSD
JTAG, microSD	IO15	5V	

Legend:

- NC/XTAL - *32.768 kHz Oscillator*
- JTAG - *JTAG / JP2*
- Boot - Boot button / SW2
- Camera - *Camera / JP4*
- LED - *RGB LED*
- microSD - *microSD Card / J4*
- LCD - *LCD / U5*
- PSRAM - ESP32-WROVER-E's PSRAM

32.768 kHz Oscillator

.	ESP32 Pin
1	GPIO32
2	GPIO33

Note: Since GPIO32 and GPIO33 are connected to the oscillator by default, they are not connected to the JP1 I/O connector to maintain signal integrity. This allocation may be changed from the oscillator to JP1 by desoldering the zero-ohm resistors from positions R11 or R23 and re-soldering them to positions R12 or R24.

SPI Flash / JP2

.	ESP32 Pin
1	CLK / GPIO6
2	SD0 / GPIO7
3	SD1 / GPIO8
4	SD2 / GPIO9
5	SD3 / GPIO10
6	CMD / GPIO11

Important: The module's flash bus is connected to the jumper block JP2 through zero-ohm resistors R140 ~ R145. If the flash memory needs to operate at the frequency of 80 MHz, for reasons such as improving the integrity of bus signals, you can desolder these resistors to disconnect the module's flash bus from the pin header JP2.

JTAG / JP2

.	ESP32 Pin	JTAG Signal
1	EN	TRST_N
2	MTMS / GPIO14	TMS
3	MTDO / GPIO15	TDO
4	MTDI / GPIO12	TDI
5	MTCK / GPIO13	TCK

Camera / JP4

.	ESP32 Pin	Camera Signal
1	n/a	3.3V
2	n/a	Ground
3	GPIO27	SIO_C / SCCB Clock
4	GPIO26	SIO_D / SCCB Data
5	GPIO25	VSYNC / Vertical Sync
6	GPIO23	HREF / Horizontal Reference
7	GPIO22	PCLK / Pixel Clock
8	GPIO21	XCLK / System Clock
9	GPIO35	D7 / Pixel Data Bit 7
10	GPIO34	D6 / Pixel Data Bit 6
11	GPIO39	D5 / Pixel Data Bit 5
12	GPIO36	D4 / Pixel Data Bit 4
13	GPIO19	D3 / Pixel Data Bit 3
14	GPIO18	D2 / Pixel Data Bit 2
15	GPIO5	D1 / Pixel Data Bit 1
16	GPIO4	D0 / Pixel Data Bit 0
17	GPIO0	RESET / Camera Reset
18	n/a	PWDN / Camera Power Down

- Signals D0 .. D7 denote camera data bus

RGB LED

.	ESP32 Pin	RGB LED
1	GPIO0	Red
2	GPIO2	Green
3	GPIO4	Blue

microSD Card

.	ESP32 Pin	microSD Signal
1	MTDI / GPIO12	DATA2
2	MTCK / GPIO13	CD / DATA3
3	MTDO / GPIO15	CMD
4	MTMS / GPIO14	CLK
5	GPIO2	DATA0
6	GPIO4	DATA1
7	GPIO21	Card Detect

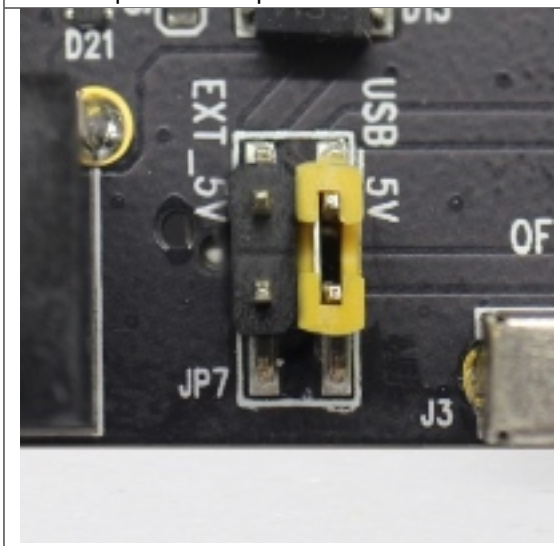
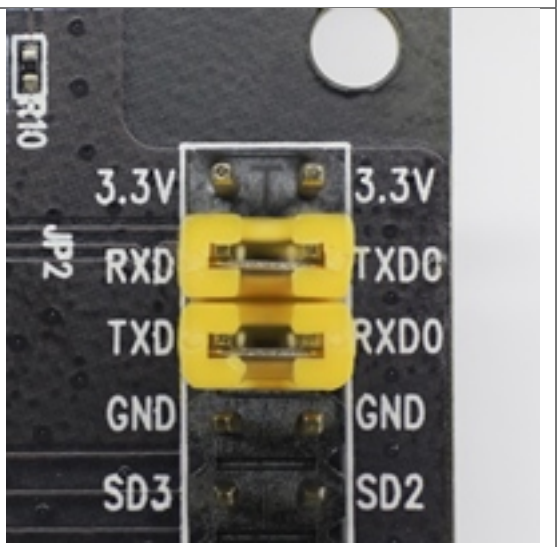
LCD / U5

.	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

Start Application Development Before powering up your ESP-WROVER-KIT, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup Please set only the following jumpers shown in the pictures below:

- Select USB as the power source using the jumper block JP7.
- Enable UART communication using the jumper block JP2.

Power up from USB port	Enable UART communication
	

Do not install any other jumpers.

Turn the **Power Switch** to ON, and the **5V Power On LED** should light up.

Now to Development Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

A Board Support Package can be found in IDF Component Registry.

The application examples that use some hardware specific to your ESP-WROVER-KIT can be found below.

- On-board LCD example: peripherals/spi_master/lcd
- SD card slot example: storage/sd_card
- Camera connector example: https://github.com/espressif/esp32-camera

Related Documents

- ESP-WROVER-KIT V4.1 schematic (PDF)
- ESP-WROVER-KIT V4.1 layout (DXF) may be opened online with Autodesk Viewer
- ESP32 Datasheet (PDF)
- ESP32-WROVER-E Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

ESP-WROVER-KIT V3 Getting Started Guide

This guide shows how to get started with the ESP-WROVER-KIT V3 development board and also provides information about its functionality and configuration options. For the description of other ESP-WROVER-KIT versions, please check *Hardware Reference*.

What You Need

- *ESP-WROVER-KIT V3 board*
- USB 2.0 cable（A to Micro-B）
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP-WROVER-KIT is an ESP32-based development board produced by Espressif. This board features an integrated LCD screen and microSD card slot.

ESP-WROVER-KIT comes with the following ESP32 modules:

- ESP32-WROOM-32
- ESP32-WROVER series

Its another distinguishing feature is the embedded FTDI FT2232HL chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface without a separate JTAG debugger. ESP-WROVER-KIT makes development convenient, easy, and cost-effective.

Most of the ESP32 I/O pins are broken out to the board's pin headers for easy access.

Note: The version with the ESP32-WROVER module uses ESP32's GPIO16 and GPIO17 as chip select and clock signals for PSRAM. By default, the two GPIOs are not broken out to the board's pin headers in order to ensure reliable performance.

Functionality Overview The block diagram below shows the main components of ESP-WROVER-KIT and their interconnections.

Functional Description The following two figures and the table below describe the key components, interfaces, and controls of the ESP-WROVER-KIT board.

The table below provides description in the following manner:

- Starting from the first picture's top right corner and going clockwise
- Then moving on to the second picture

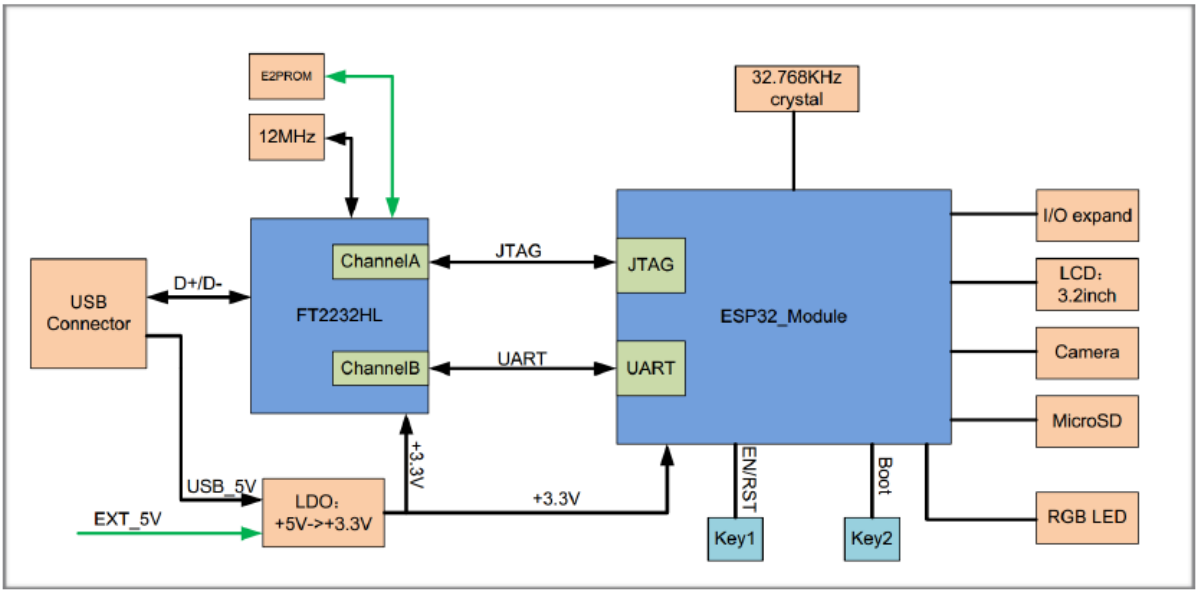


Fig. 9: ESP-WROVER-KIT block diagram

Key Component	Description
32.768 kHz	External precision 32.768 kHz crystal oscillator serves as a clock with low-power consumption while the chip is in Deep-sleep mode.
0R	Zero-ohm resistor intended as a placeholder for a current shunt, can be desoldered or replaced with a current shunt to facilitate the measurement of ESP32's current consumption in different modes.
ESP32 Module	Either ESP32-WROOM-32 or ESP32-WROVER with an integrated ESP32. The ESP32-WROVER module features all the functions of ESP32-WROOM-32 and integrates an external 32-MBit PSRAM for flexible extended storage and data processing capabilities.
FT2232	The FT2232 chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232 also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232 chip enhances user-friendliness in terms of application development and debugging. See ESP-WROVER-KIT V3 schematic.
UART	Serial port. The serial TX/RX signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP11 respectively. By default, these pairs of pins are connected with jumpers. To use ESP32's serial interface, remove the jumpers and connect another external serial device to the respective pins.
SPI	By default, ESP32 uses its SPI interface to access flash and PSRAM memory inside the module. Use these pins to connect ESP32 to another SPI device. In this case, an extra chip select (CS) signal is needed. Please note that the interface voltage for the version with ESP32-WROVER is 1.8V, while that for the version with ESP32-WROOM-32 is 3.3V.
CTS/RTS	Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, short the respective pins of JP14 with jumpers.
JTAG	JTAG interface. JTAG signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP8 respectively. By default, these pairs of pins are disconnected. To enable JTAG, short the respective pins with jumpers as shown in Section *Setup Options*.
EN	Reset button.
Boot	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
USB	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power Key	Power On/Off Switch. Toggling toward **USB** powers the board on, toggling away from **USB** powers the board off.
Power Select	Power supply selector interface. The board can be powered either via USB or via the 5V Input interface. Select the power source with a jumper. For more details, see Section *Setup Options*, jumper header JP7.

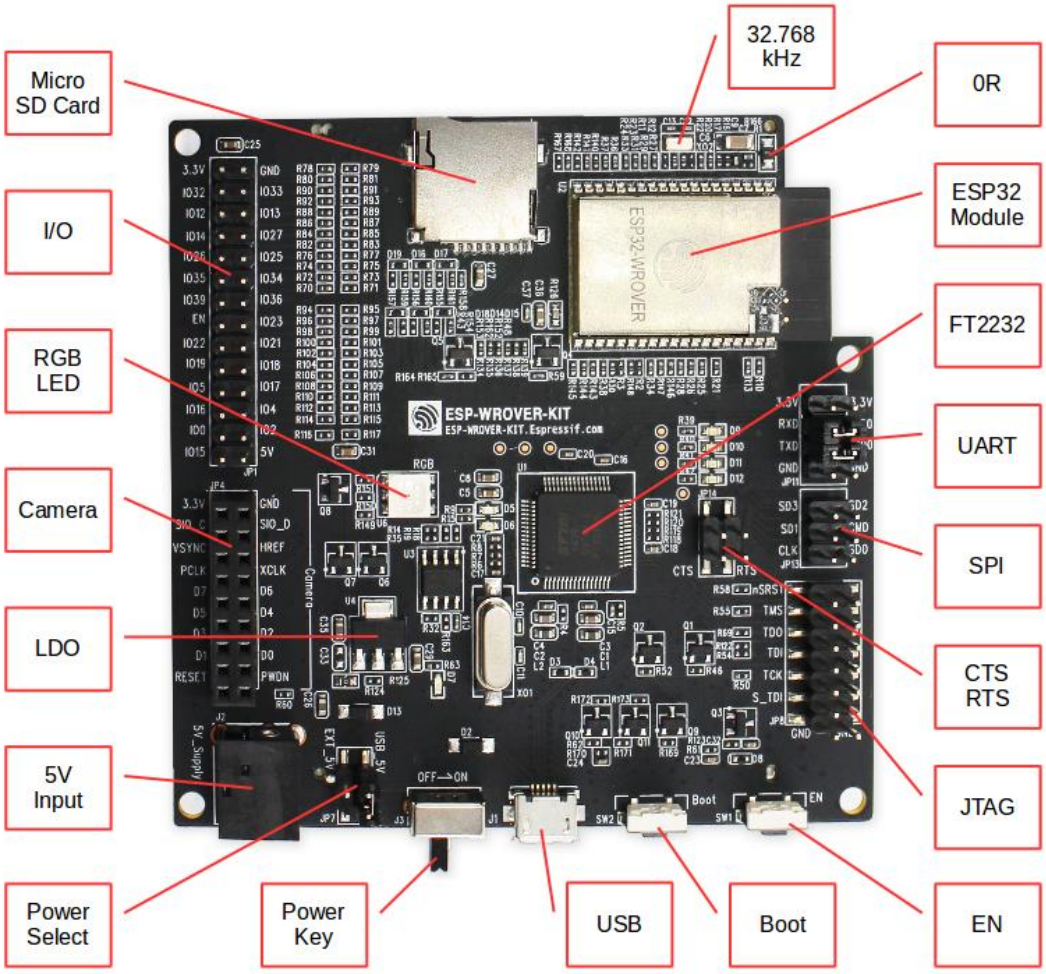


Fig. 10: ESP-WROVER-KIT board layout - front

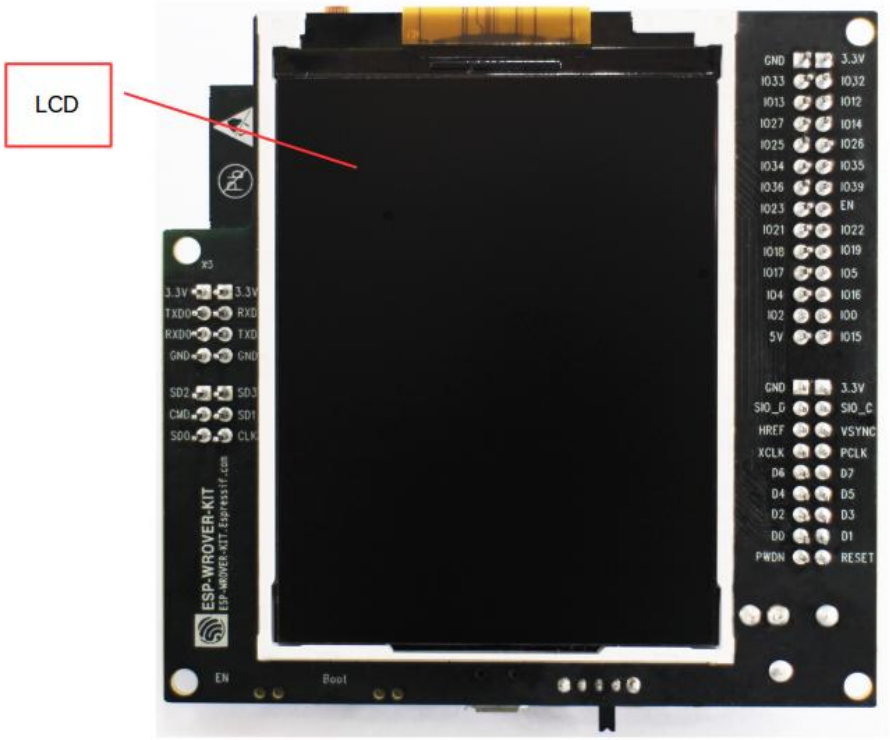
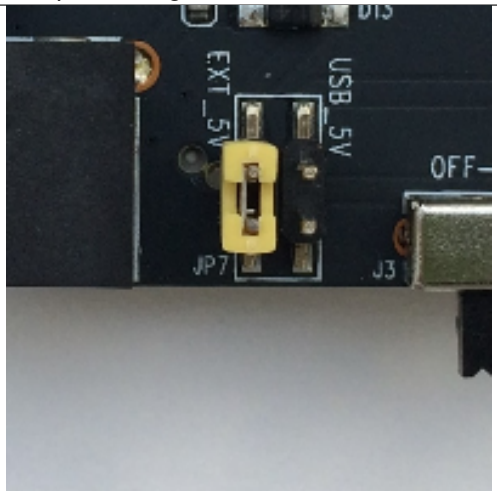
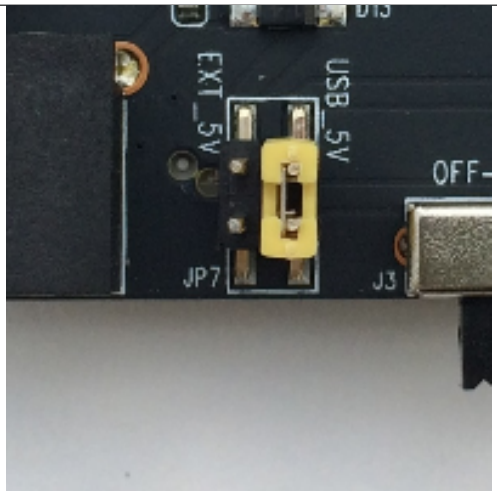
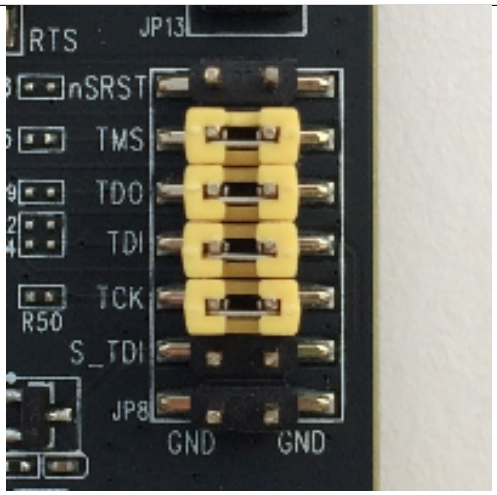
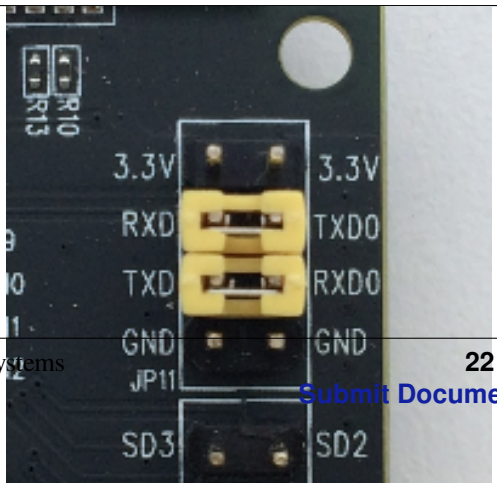


Fig. 11: ESP-WROVER-KIT board layout - back

Setup Options There are five jumper blocks available to set up the board functionality. The most frequently required options are listed in the table below.

Header	Jumper Setting	Description of Functionality
JP7		Power ESP-WROVER-KIT via an external power supply
JP7		Power ESP-WROVER-KIT via USB
JP8		Enable JTAG functionality
JP11		Enable UART communication

Allocation of ESP32 Pins Some pins / terminals of ESP32 are allocated for use with the onboard or external hardware. If that hardware is not used, e.g., nothing is plugged into the Camera (JP4) header, then these GPIOs can be used for other purposes.

Some of the pins, such as GPIO0 or GPIO2, have multiple functions and some of them are shared among onboard and external peripheral devices. Certain combinations of peripherals cannot work together. For example, it is not possible to do JTAG debugging of an application that is using SD card, because several pins are shared by JTAG and the SD card slot.

In other cases, peripherals can coexist under certain conditions. This is applicable to, for example, LCD screen and SD card that share only a single pin GPIO21. This pin is used to provide D/C (Data / Control) signal for the LCD as well as the CD (Card Detect) signal read from the SD card slot. If the card detect functionality is not essential, then it may be disabled by removing R167, so both LCD and SD may operate together.

For more details on which pins are shared among which peripherals, please refer to the table in the next section.

Main I/O Connector / JP1 The JP1 connector consists of 14x2 male pins whose functions are shown in the middle two "I/O" columns of the table below. The two "Shared With" columns on both sides describe where else on the board a certain GPIO is used.

Shared With	I/O	I/O	Shared With
n/a	3.3V	GND	n/a
NC/XTAL	IO32	IO33	NC/XTAL
JTAG, microSD	IO12	IO13	JTAG, microSD
JTAG, microSD	IO14	IO27	Camera
Camera	IO26	IO25	Camera, LCD
Camera	IO35	IO34	Camera
Camera	IO39	IO36	Camera
JTAG	EN	IO23	Camera, LCD
Camera, LCD	IO22	IO21	Camera, LCD, microSD
Camera, LCD	IO19	IO18	Camera, LCD
Camera, LCD	IO5	IO17	PSRAM
PSRAM	IO16	IO4	LED, Camera, microSD
Camera, LED, Boot	IO0	IO2	LED, microSD
JTAG, microSD	IO15	5V	

Legend:

- NC/XTAL - *32.768 kHz Oscillator*
- JTAG - *JTAG / JP8*
- Boot - Boot button / SW2
- Camera - *Camera / JP4*
- LED - *RGB LED*
- microSD - *microSD Card / J4*
- LCD - *LCD / U5*
- PSRAM - only in case ESP32-WROVER is installed

32.768 kHz Oscillator

.	ESP32 Pin
1	GPIO32
2	GPIO33

Note: Since GPIO32 and GPIO33 are connected to the oscillator by default, they are not connected to the JP1 I/O connector to maintain signal integrity. This allocation may be changed from the oscillator to JP1 by desoldering the zero-ohm resistors from positions R11 / R23 and re-soldering them to positions R12 / R24.

SPI Flash / JP13

.	ESP32 Pin
1	CLK / GPIO6
2	SD0 / GPIO7
3	SD1 / GPIO8
4	SD2 / GPIO9
5	SD3 / GPIO10
6	CMD / GPIO11

Important: The module's flash bus is connected to the jumper block JP13 through zero-ohm resistors R140 ~ R145. If the flash memory needs to operate at the frequency of 80 MHz, for reasons such as improving the integrity of bus signals, you can desolder these resistors to disconnect the module's flash bus from the pin header JP13.

JTAG / JP8

.	ESP32 Pin	JTAG Signal
1	EN	TRST_N
2	MTMS / GPIO14	TMS
3	MTDO / GPIO15	TDO
4	MTDI / GPIO12	TDI
5	MTCK / GPIO13	TCK

Camera / JP4

.	ESP32 Pin	Camera Signal
1	n/a	3.3V
2	n/a	Ground
3	GPIO27	SIO_C / SCCB Clock
4	GPIO26	SIO_D / SCCB Data
5	GPIO25	VSYNC / Vertical Sync
6	GPIO23	HREF / Horizontal Reference
7	GPIO22	PCLK / Pixel Clock
8	GPIO21	XCLK / System Clock
9	GPIO35	D7 / Pixel Data Bit 7
10	GPIO34	D6 / Pixel Data Bit 6
11	GPIO39	D5 / Pixel Data Bit 5
12	GPIO36	D4 / Pixel Data Bit 4
13	GPIO19	D3 / Pixel Data Bit 3
14	GPIO18	D2 / Pixel Data Bit 2
15	GPIO5	D1 / Pixel Data Bit 1
16	GPIO4	D0 / Pixel Data Bit 0
17	GPIO0	RESET / Camera Reset
18	n/a	PWDN / Camera Power Down

- Signals D0 .. D7 denote camera data bus

RGB LED

.	ESP32 Pin	RGB LED
1	GPIO0	Red
2	GPIO2	Green
3	GPIO4	Blue

microSD Card

.	ESP32 Pin	microSD Signal
1	MTDI / GPIO12	DATA2
2	MTCK / GPIO13	CD / DATA3
3	MTDO / GPIO15	CMD
4	MTMS / GPIO14	CLK
5	GPIO2	DATA0
6	GPIO4	DATA1
7	GPIO21	CD

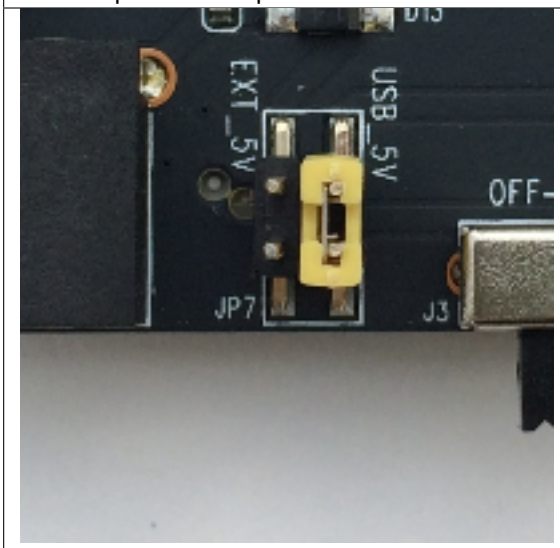
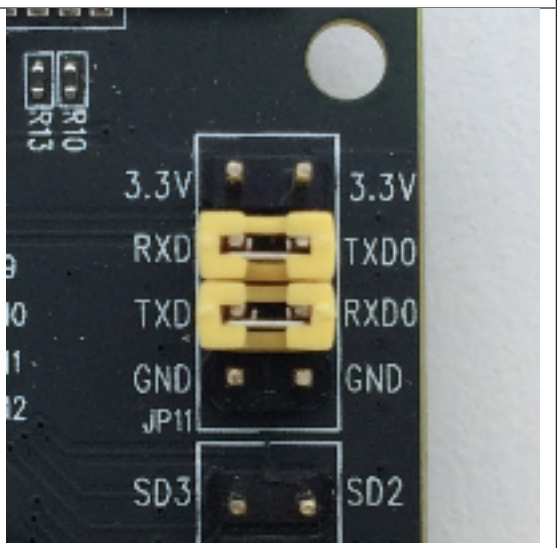
LCD / U5

.	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

Start Application Development Before powering up your ESP-WROVER-KIT, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup Please set only the following jumpers shown in the pictures below:

- Select USB as the power source using the jumper block JP7.
- Enable UART communication using the jumper block JP11.

Power up from USB port	Enable UART communication
	

Do not install any other jumpers.

Turn the **Power Switch** to ON, the **5V Power On LED** should light up.

Now to Development Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- ESP-WROVER-KIT V3 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- ESP32-WROOM-32 Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

ESP-WROVER-KIT V2 Getting Started Guide

This guide shows how to get started with the ESP-WROVER-KIT V2 development board and also provides information about its functionality and configuration options. For the description of other ESP-WROVER-KIT versions, please check *Hardware Reference*.

What You Need

- ESP-WROVER-KIT V2 board
- USB 2.0 cable（A to Micro-B）
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP-WROVER-KIT is an ESP32-based development board produced by Espressif. This board features an integrated LCD screen and microSD card slot.

ESP-WROVER-KIT comes with the following ESP32 modules:

- ESP32-WROOM-32
- ESP32-WROVER series

Its another distinguishing feature is the embedded FTDI FT2232HL chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface without a separate JTAG debugger. ESP-WROVER-KIT makes development convenient, easy, and cost-effective.

Most of the ESP32 I/O pins are broken out to the board's pin headers for easy access.

Note: The version with the ESP32-WROVER module uses ESP32's GPIO16 and GPIO17 as chip select and clock signals for PSRAM. By default, the two GPIOs are not broken out to the board's pin headers in order to ensure reliable performance.

Functionality Overview The block diagram below shows the main components of ESP-WROVER-KIT and their interconnections.

Functional Description The following two figures and the table below describe the key components, interfaces, and controls of the ESP-WROVER-KIT board.

The table below provides description in the following manner:

- Starting from the first picture's top right corner and going clockwise
- Then moving on to the second picture

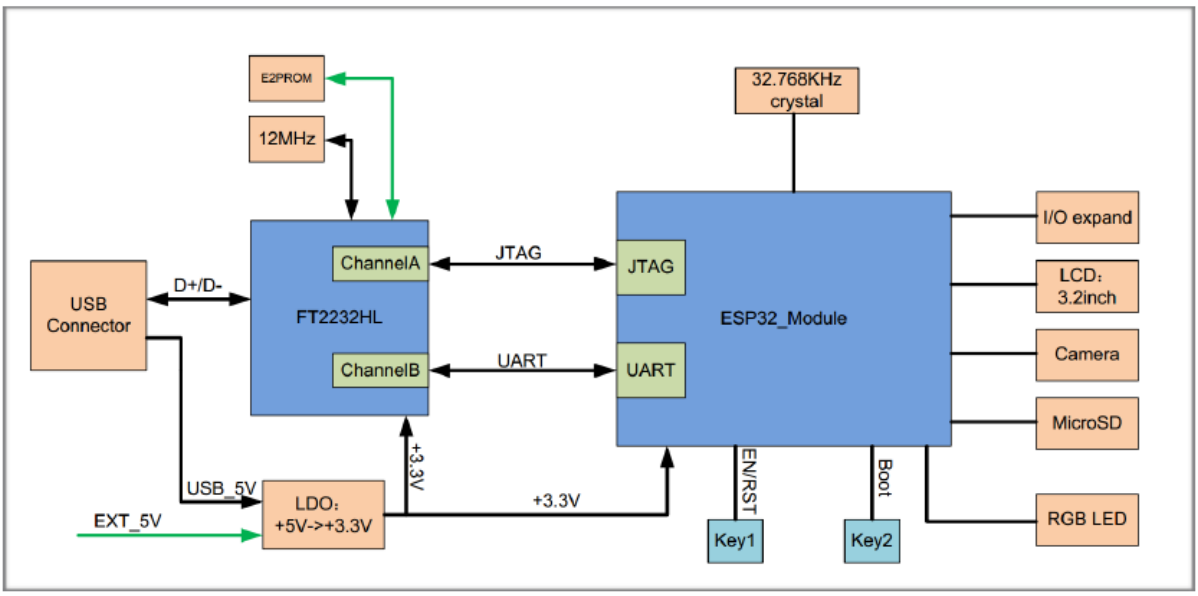


Fig. 12: ESP-WROVER-KIT block diagram

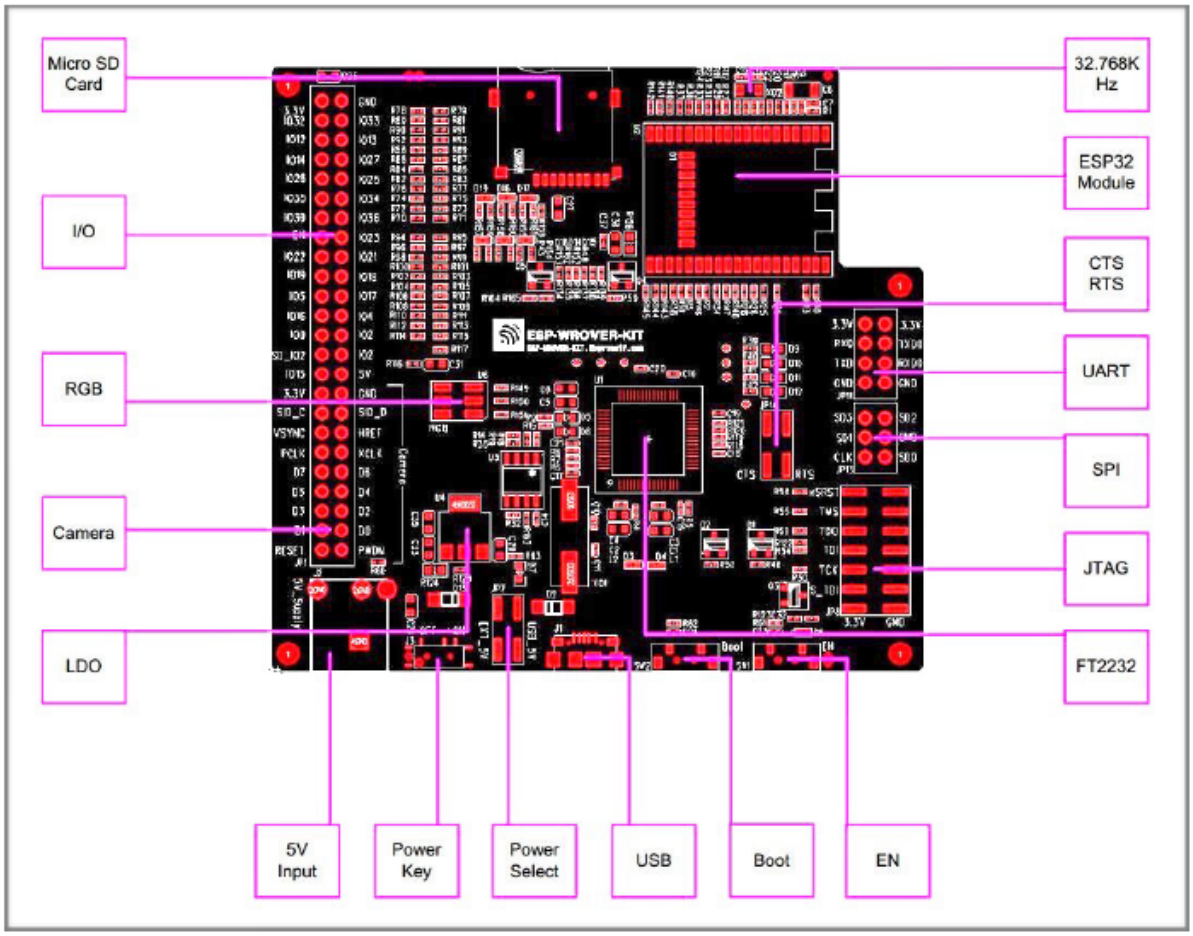


Fig. 13: ESP-WROVER-KIT board layout - front

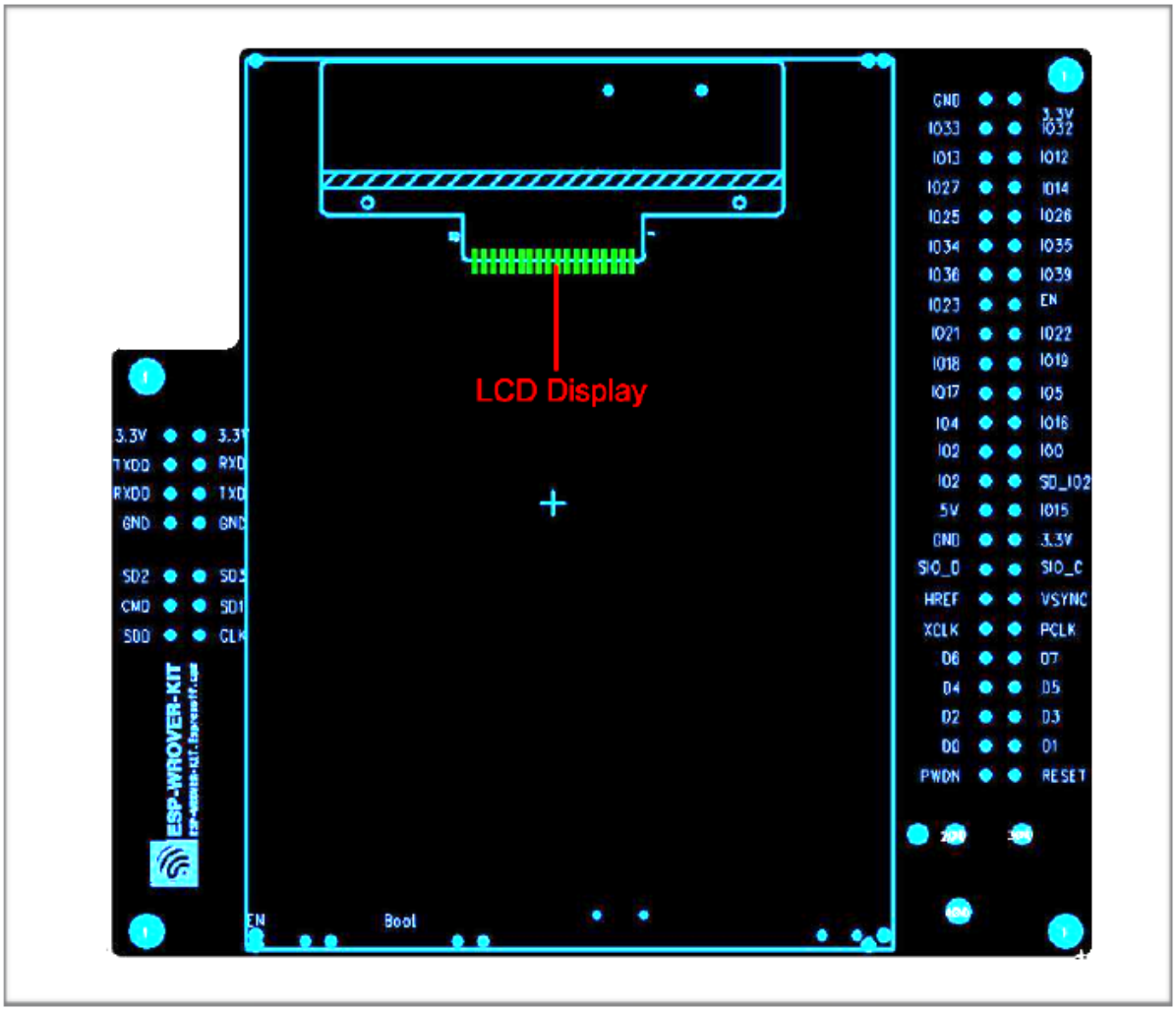
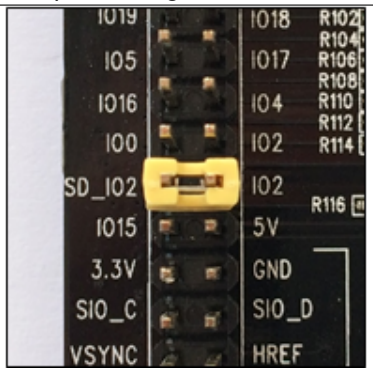
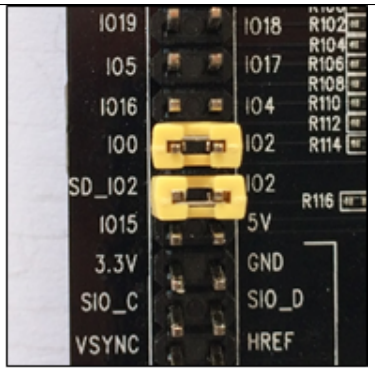
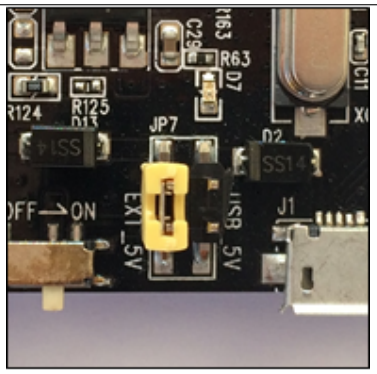
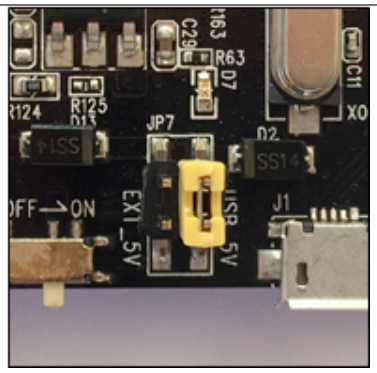
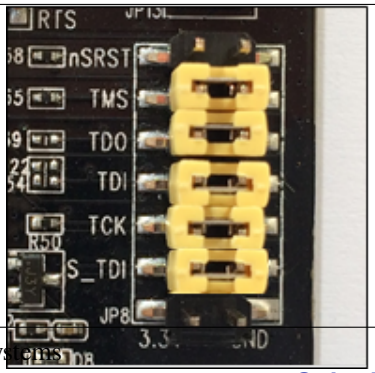
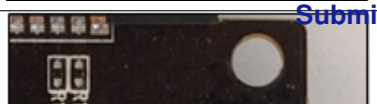


Fig. 14: ESP-WROVER-KIT board layout - back

Key Component	Description
32.768 kHz	External precision 32.768 kHz crystal oscillator serves as a clock with low-power consumption while the chip is in Deep-sleep mode.
ESP32 Module	Either ESP32-WROOM-32 or ESP32-WROVER with an integrated ESP32. The ESP32-WROVER module features all the functions of ESP32-WROOM-32 and integrates an external 32-MBit PSRAM for flexible extended storage and data processing capabilities.
CTS/RTS	Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, short the respective pins of JP14 with jumpers.
UART	Serial port. The serial TX/RX signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP11 respectively. By default, these pairs of pins are connected with jumpers. To use ESP32 ' s serial interface, remove the jumpers and connect another external serial device to the respective pins.
SPI	By default, ESP32 uses its SPI interface to access flash and PSRAM memory inside the module. Use these pins to connect ESP32 to another SPI device. In this case, an extra chip select (CS) signal is needed. Please note that the interface voltage for the version with ESP32-WROVER is 1.8V, while that for the version with ESP32-WROOM-32 is 3.3 V.
JTAG	JTAG interface. JTAG signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP8 respectively. By default, these pairs of pins are disconnected. To enable JTAG, short the respective pins with jumpers as shown in Section *Setup Options*.
FT2232	The FT2232 chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232 features USB-to-UART and USB-to-JTAG functionalities.
EN	Reset button.
Boot	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
USB	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power Select	Power supply selector interface. The board can be powered either via USB or via the 5 V Input interface. Select the power source with a jumper. For more details, see Section *Setup Options*, jumper header JP7.
Power Key	Power On/Off Switch. Toggling toward **USB** powers the board on, toggling away from **USB** powers the board off.
5V Input	The 5 V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer).
LDO	NCP1117(1 A). 5V-to-3.3V LDO. NCP1117 can provide a maximum current of 1 A. The LDO on the board has a fixed output voltage. Although, the user can install an LDO with adjustable output voltage. For details, please refer to ESP-WROVER-KIT V2 schematic.
Camera	Camera interface, a standard OV7670 camera module.
RGB	Red, green and blue (RGB) light emitting diodes (LEDs), can be controlled by pulse width modulation (PWM).
I/O	All the pins on the ESP32 module are broken out to pin headers. You can program ESP32 to enable multiple functions, such as PWM, ADC, DAC, I2C, I2S, SPI, etc.
microSD Card	microSD card slot for data storage: when ESP32 enters the download mode, GPIO2 cannot be held high. However, a pull-up resistor is required on GPIO2 to enable the microSD Card. By default, GPIO2 and the pull-up resistor R153 are disconnected. To enable the SD Card, use jumpers on JP1 as shown in Section *Setup Options*.
LCD	Support for mounting and interfacing a 3.2" SPI (standard 4-wire Serial Peripheral Interface) LCD, as shown on figure *ESP-WROVER-KIT board layout - back*.

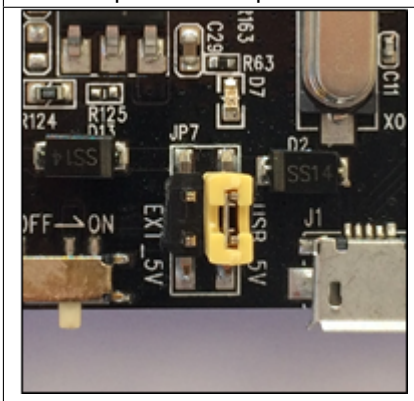
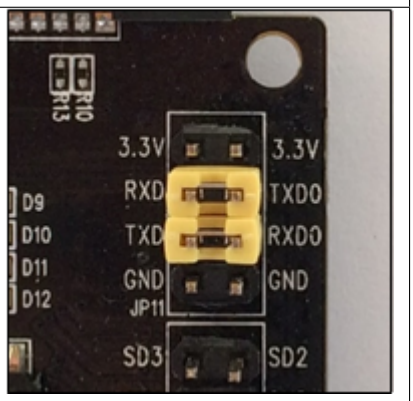
Setup Options There are five jumper blocks available to set up the board functionality. The most frequently required options are listed in the table below.

Header	Jumper Setting	Description of Functionality
JP1		Enable pull up for the microSD Card
JP1		Assert GPIO2 low during each download (by jumping it to GPIO0)
JP7		Power ESP-WROVER-KIT via an external power supply
JP7		Power ESP-WROVER-KIT via USB
JP8		Enable JTAG functionality
JP8		

Start Application Development Before powering up your ESP-WROVER-KIT, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup Please set only the following jumpers shown in the pictures below:

- Select USB as the power source using the jumper block JP7.
- Enable UART communication using the jumper block JP11.

Power up from USB port	Enable UART communication
	

Do not install any other jumpers.

Turn the **Power Switch** to ON, the **5V Power On LED** should light up.

Now to Development Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- ESP-WROVER-KIT V2 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- ESP32-WROOM-32 Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

ESP32-PICO-KIT V4 / V4.1 Getting Started Guide

This guide shows how to get started with the ESP32-PICO-KIT V4/V4.1 mini development board. For the description of other ESP32-PICO-KIT versions, please check *Hardware Reference*.

This particular description covers ESP32-PICO-KIT V4 and V4.1. The difference is the upgraded USB-UART bridge from CP2102 in V4 with up to 1 Mbps transfer rates to CP2102N in V4.1 with up to 3 Mbps transfer rates.

What You Need

- *ESP32-PICO-KIT mini development board*
- USB 2.0 A to Micro B cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-PICO-KIT is an ESP32-based mini development board produced by Espressif.

The core of this board is ESP32-PICO-D4 - a System-in-Package (SiP) module with complete Wi-Fi and Bluetooth functionalities. Compared to other ESP32 modules, ESP32-PICO-D4 integrates the following peripheral components in one single package, which otherwise would need to be installed separately:

- 40 MHz crystal oscillator
- 4 MB flash
- Filter capacitors
- RF matching links

This setup reduces the costs of additional external components as well as the cost of assembly and testing and also increases the overall usability of the product.

The development board features a USB-UART Bridge circuit which allows developers to connect the board to a computer's USB port for flashing and debugging.

All the IO signals and system power on ESP32-PICO-D4 are led out to two rows of 20 x 0.1" header pads on both sides of the development board for easy access. For compatibility with Dupont wires, 2 x 17 header pads are populated with two rows of male pin headers. The remaining 2 x 3 header pads beside the antenna are not populated. These pads may be populated later by the user if required.

Note:

1. There are two versions of ESP32-PICO-KIT boards, respectively with male headers and female headers. In this guide, the male header version is taken as an example.
2. The 2 x 3 pads not populated with pin headers are connected to the flash memory embedded in the ESP32-PICO-D4 SiP module. For more details, see module's datasheet in *Related Documents*.

Functionality Overview The block diagram below shows the main components of ESP32-PICO-KIT and their interconnections.

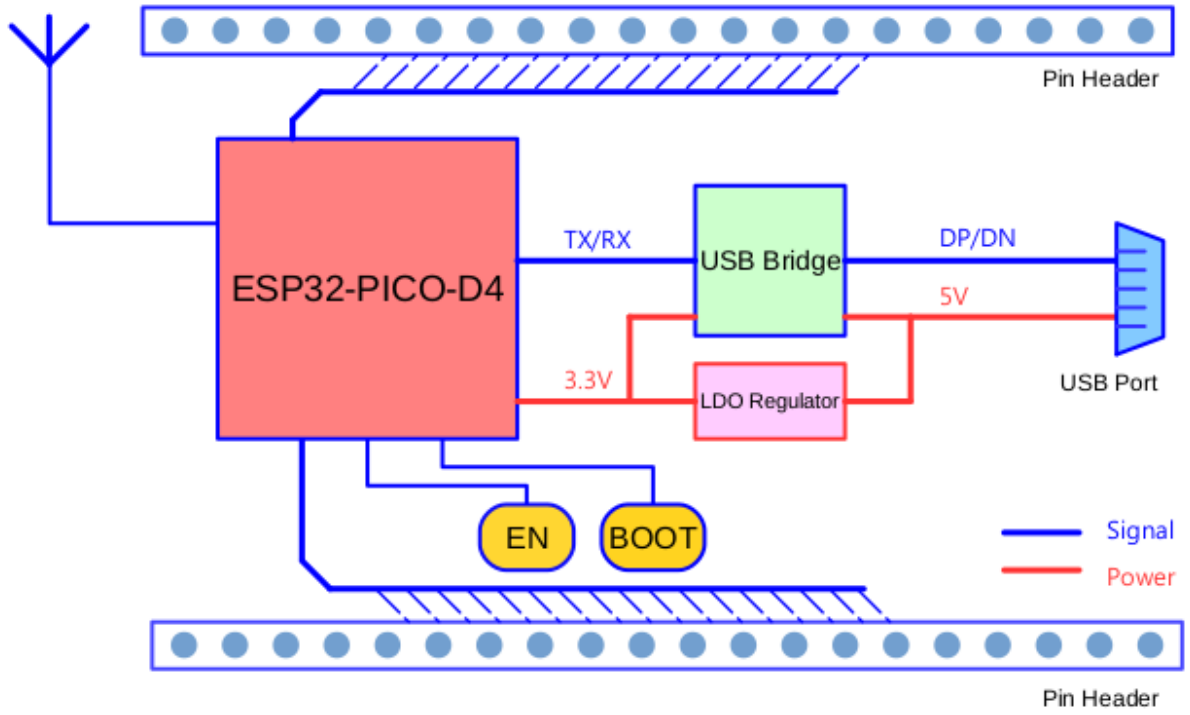


Fig. 15: ESP32-PICO-KIT block diagram

Functional Description The following figure and the table below describe the key components, interfaces, and controls of the ESP32-PICO-KIT board.

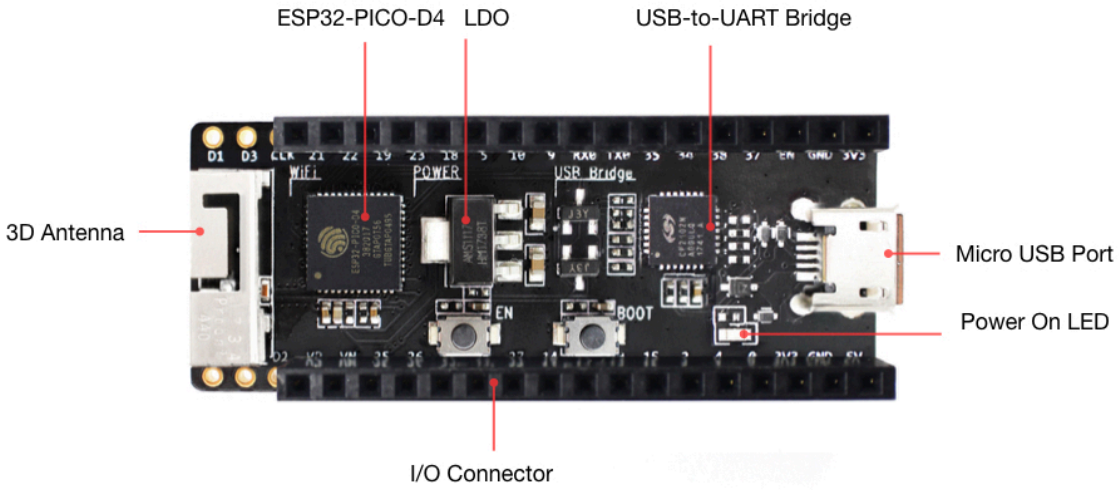


Fig. 16: ESP32-PICO-KIT board layout (with female headers)

Below is the description of the items identified in the figure starting from the top left corner and going clockwise.

Key Component	Description
ESP32-PICO-D4	Standard ESP32-PICO-D4 module soldered to the ESP32-PICO-KIT board. The complete ESP32 system on a chip (ESP32 SoC) has been integrated into the SiP module, requiring only an external antenna with LC matching network, decoupling capacitors, and a pull-up resistor for EN signals to function properly.
LDO	5V-to-3.3V Low dropout voltage regulator (LDO).
USB-UART bridge	Single-chip USB-UART bridge: CP2102 in V4 provides up to 1 Mbps transfer rates and CP2102N in V4.1 offers up to 3 Mbps transfers rates.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
5V Power On LED	This red LED turns on when power is supplied to the board. For details, see the schematics in *Related Documents*.
I/O	All the pins on ESP32-PICO-D4 are broken out to pin headers. You can program ESP32 to enable multiple functions, such as PWM, ADC, DAC, I2C, I2S, SPI, etc. For details, please see Section *Pin Descriptions*.
BOOT Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
EN Button	Reset button.

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V / GND header pins
- 3V3 / GND header pins

Warning: The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.

Pin Descriptions The two tables below provide the **Name** and **Function** of I/O header pins on both sides of the board, see *ESP32-PICO-KIT board layout (with female headers)*. The pin numbering and header names are the same as in the schematic given in *Related Documents*.

Header J2

No.	Name	Type	Function
1	FLASH_SD1 (FSD1)	I/O	GPIO8, SD_DATA1, SPID, HS1_DATA1 *(See 1)* , U2CTS
2	FLASH_SD3 (FSD3)	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0 *(See 1)* , U2RTS
3	FLASH_CLK (FCLK)	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK *(See 1)* , U1CTS
4	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
5	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
6	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
7	IO23	I/O	GPIO23, VSPID, HS1_STROBE
8	IO18	I/O	GPIO18, VSPICLK, HS1_DATA7
9	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
10	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
11	IO9	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD
12	RXD0	I/O	GPIO3, U0RXD *(See 3)* , CLK_OUT2
13	TXD0	I/O	

Header J3

No.	Name	Type	Function
1	FLASH_CS (FCS)	I/O	GPIO16, HS1_DATA4 *(See 1)* , U2RXD, EMAC_CLK_OUT
2	FLASH_SD0 (FSD0)	I/O	GPIO17, HS1_DATA5 *(See 1)* , U2TXD, EMAC_CLK_OUT_180
3	FLASH_SD2 (FSD2)	I/O	GPIO11, SD_CMD, SPICS0, HS1_CMD *(See 1)* , U1RTS
4	SENSOR_VP (FSVP)	I	GPIO36, ADC1_CH0, RTC_GPIO0
5	SENSOR_VN (FSVN)	I	GPIO39, ADC1_CH3, RTC_GPIO3
6	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
7	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
8	IO32	I/O	32K_XP *(See 2a)* , ADC1_CH4, TOUCH9, RTC_GPIO9
9	IO33	I/O	32K_XN *(See 2b)* , ADC1_CH5, TOUCH8, RTC_GPIO8
10	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17 EMAC_RX_DV
11	IO14	I/O	ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK,

Note:

1. This pin is connected to the flash pin of ESP32-PICO-D4.
2. 32.768 kHz crystal oscillator: a) input, b) output.
3. This pin is connected to the pin of the USB bridge chip on the board.
4. The operating voltage of ESP32-PICO-KIT' s embedded SPI flash is 3.3 V. Therefore, the strapping pin MTDI should hold bit zero during the module power-on reset. If connected, please make sure that this pin is not held up on reset.

Start Application Development Before powering up your ESP32-PICO-KIT, please make sure that the board is in good condition with no obvious signs of damage.

After that, proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Board Dimensions The dimensions are 52 x 20.3 x 10 mm (2.1" x 0.8" x 0.4").

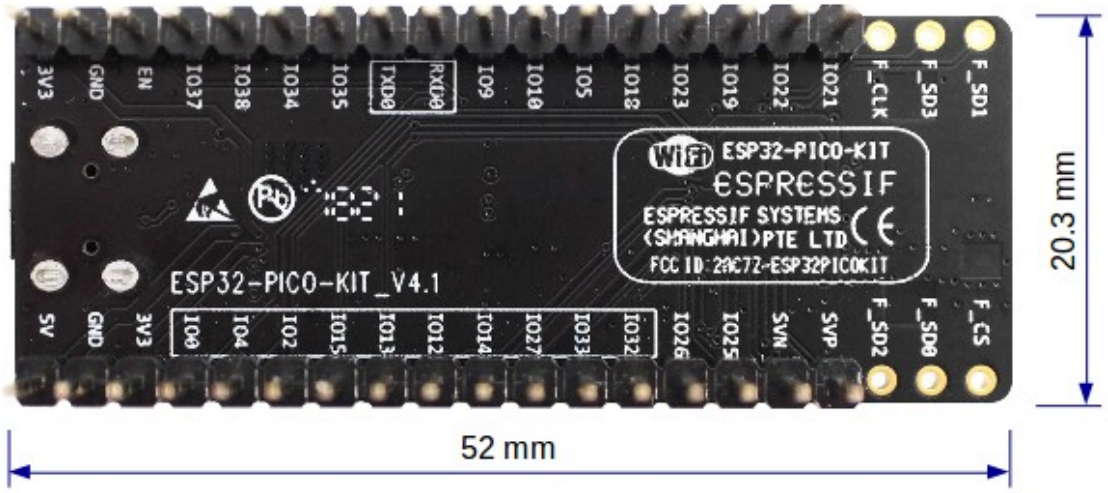


Fig. 17: ESP32-PICO-KIT dimensions - back (with male headers)

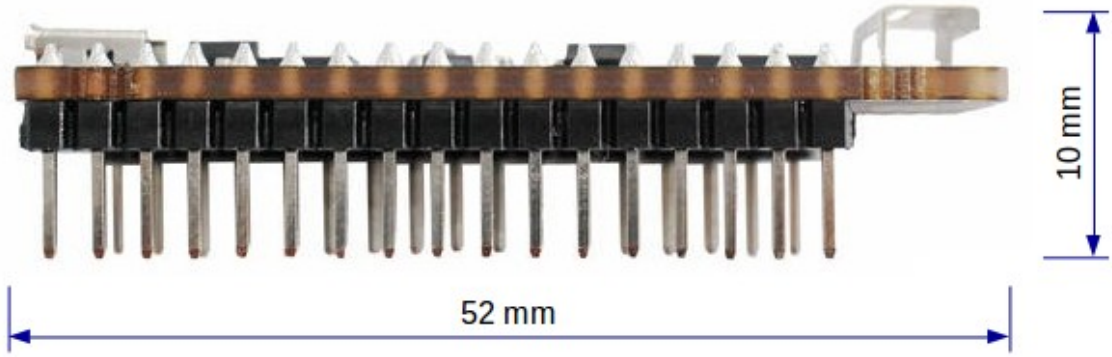


Fig. 18: ESP32-PICO-KIT dimensions - side (with male headers)

For the board physical construction details, please refer to its Reference Design listed below.

Related Documents

- ESP32-PICO-KIT V4 schematic (PDF)
- ESP32-PICO-KIT V4.1 schematic (PDF)
- ESP32-PICO-KIT Reference Design containing OrCAD schematic, PCB layout, gerbers and BOM
- ESP32-PICO-D4 Datasheet (PDF)
- *Hardware Reference*

ESP32-PICO-KIT V3 Getting Started Guide

This guide shows how to get started with the ESP32-PICO-KIT V3 mini development board. For the description of other ESP32-PICO-KIT versions, please check *Hardware Reference*.

What You Need

- ESP32-PICO-KIT V3 mini development board
- USB 2.0 A to Micro B cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-PICO-KIT V3 is an ESP32-based mini development board produced by Espressif. The core of this board is ESP32-PICO-D4 - a System-in-Package (SiP) module.

The development board features a USB-UART Bridge circuit, which allows developers to connect the board to a computer’ s USB port for flashing and debugging.

All the IO signals and system power on ESP32-PICO-D4 are led out to two rows of 20 x 0.1” header pads on both sides of the development board for easy access.

Functional Description The following figure and the table below describe the key components, interfaces, and controls of the ESP32-PICO-KIT V3 board.

Below is the description of the items identified in the figure starting from the top left corner and going clockwise.

Key Component	Description
ESP32-PICO-D4	Standard ESP32-PICO-D4 module soldered to the ESP32-PICO-KIT V3 board. The complete ESP32 system on a chip (ESP32 SoC) has been integrated into the SiP module, requiring only an external antenna with LC matching network, decoupling capacitors, and a pull-up resistor for EN signals to function properly.
LDO	5V-to-3.3V Low dropout voltage regulator (LDO).
USB-UART bridge	Single-chip USB-UART bridge provides up to 1 Mbps transfers rates.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power On LED	This red LED turns on when power is supplied to the board.
I/O	All the pins on ESP32-PICO-D4 are broken out to pin headers. You can program ESP32 to enable multiple functions, such as PWM, ADC, DAC, I2C, I2S, SPI, etc.
BOOT Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
EN Button	Reset button.

Start Application Development Before powering up your ESP32-PICO-KIT V3, please make sure that the board is in good condition with no obvious signs of damage.

After that, proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

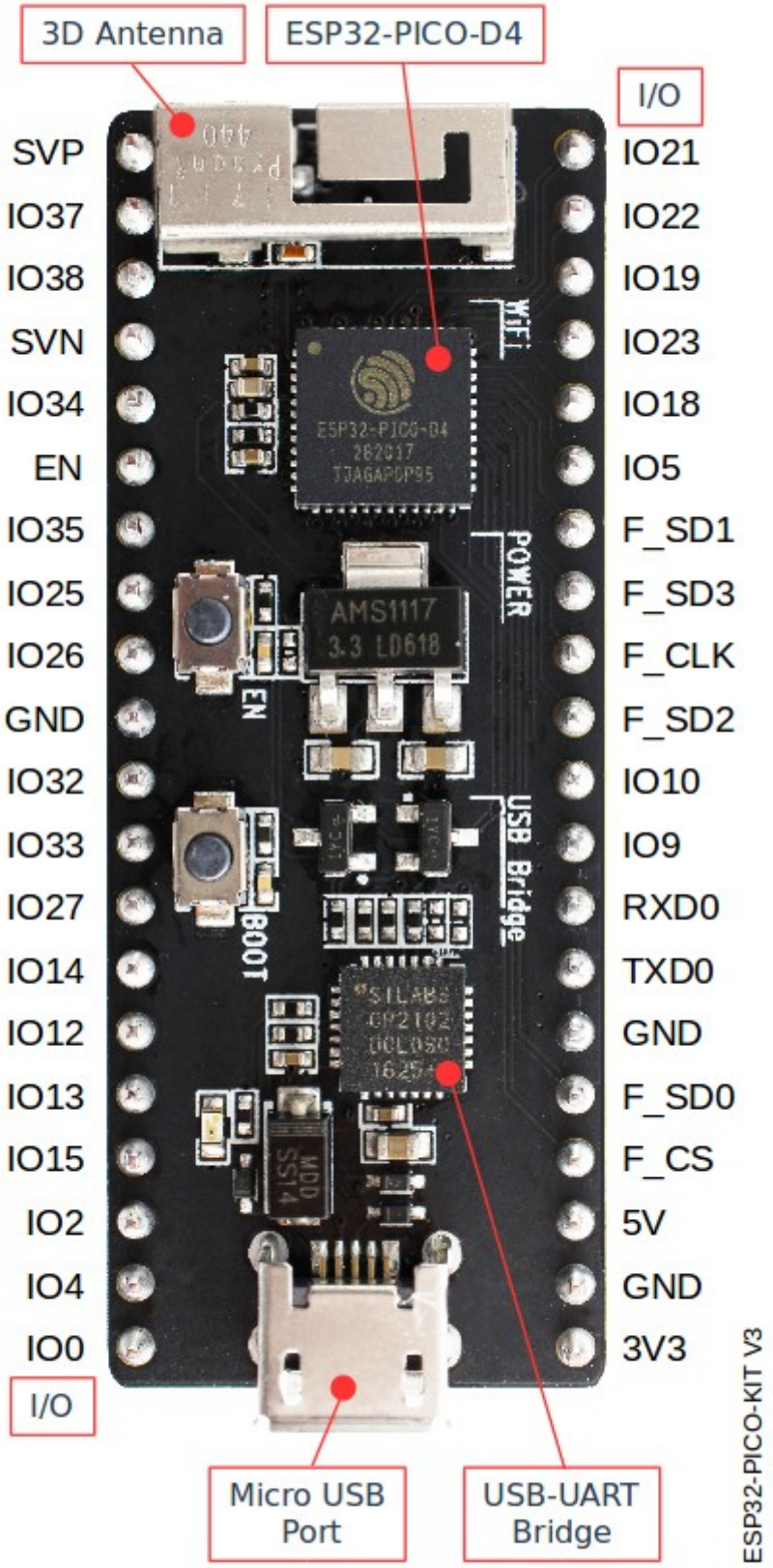


Fig. 19: ESP32-PICO-KIT V3 board layout

Related Documents

- ESP32-PICO-KIT V3 schematic (PDF)
- ESP32-PICO-D4 Datasheet (PDF)
- *Hardware Reference*

ESP32-Ethernet-Kit V1.2 Getting Started Guide

This guide shows how to get started with the ESP32-Ethernet-Kit development board and also provides information about its functionality and configuration options.

The *ESP32-Ethernet-Kit* is an Ethernet-to-Wi-Fi development board that enables Ethernet devices to be interconnected over Wi-Fi. At the same time, to provide more flexible power supply options, the ESP32-Ethernet-Kit also supports power over Ethernet (PoE).

Fig. 20: ESP32-Ethernet-Kit V1.2 Overview (click to enlarge)

What You Need

- *ESP32-Ethernet-Kit V1.2 board*
- USB 2.0 A to Micro B Cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-Ethernet-Kit is an ESP32-based development board produced by Espressif.

It consists of two development boards, the Ethernet board A and the PoE board B. The *Ethernet board (A)* contains Bluetooth/Wi-Fi dual-mode ESP32-WROVER-E module and IP101GRI, a Single Port 10/100 Fast Ethernet Transceiver (PHY). The *PoE board (B)* provides power over Ethernet functionality. The A board can work independently, without the board B installed.

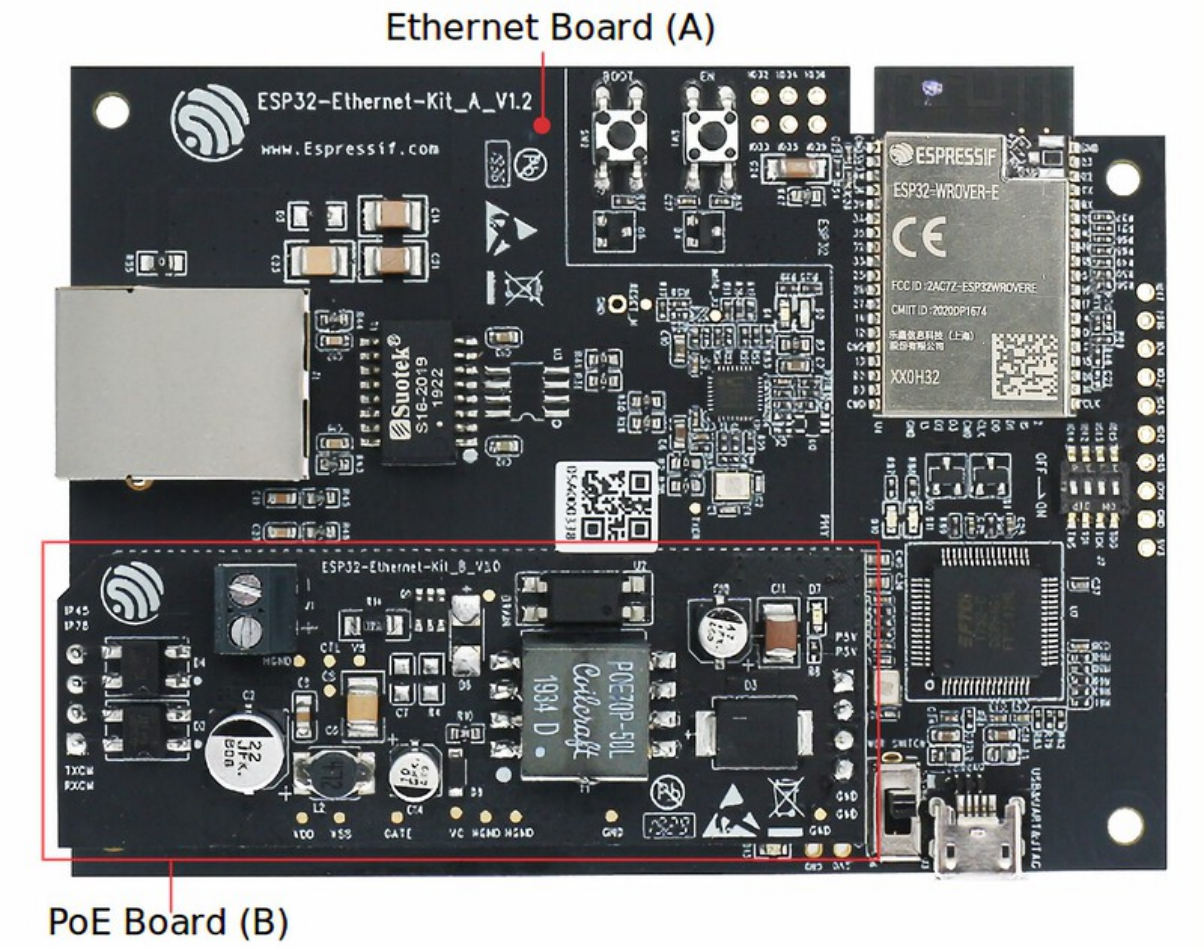


Fig. 21: ESP32-Ethernet-Kit V1.2 (click to enlarge)

For the application loading and monitoring, the Ethernet board (A) also features FTDI FT2232H chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface without a separate JTAG debugger.

Functionality Overview The block diagram below shows the main components of ESP32-Ethernet-Kit and their interconnections.

Functional Description The following figures and tables describe the key components, interfaces, and controls of the ESP32-Ethernet-Kit.

Ethernet Board (A) The table below provides description starting from the picture's top right corner and going clockwise.

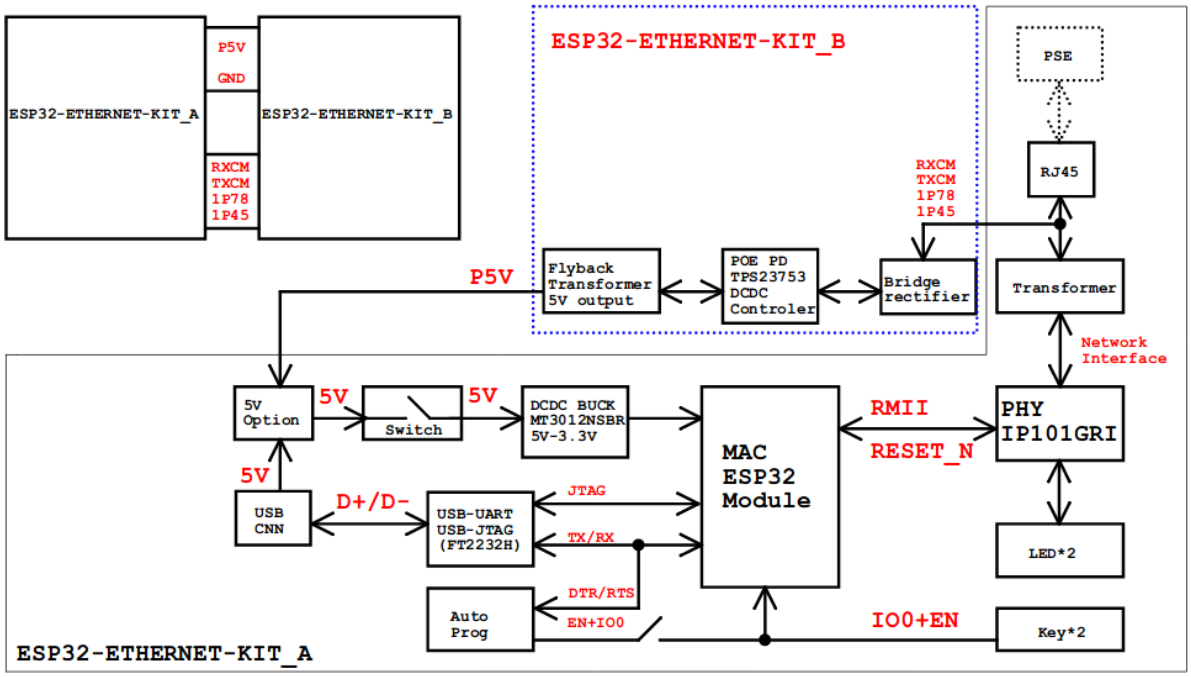


Fig. 22: ESP32-Ethernet-Kit block diagram (click to enlarge)

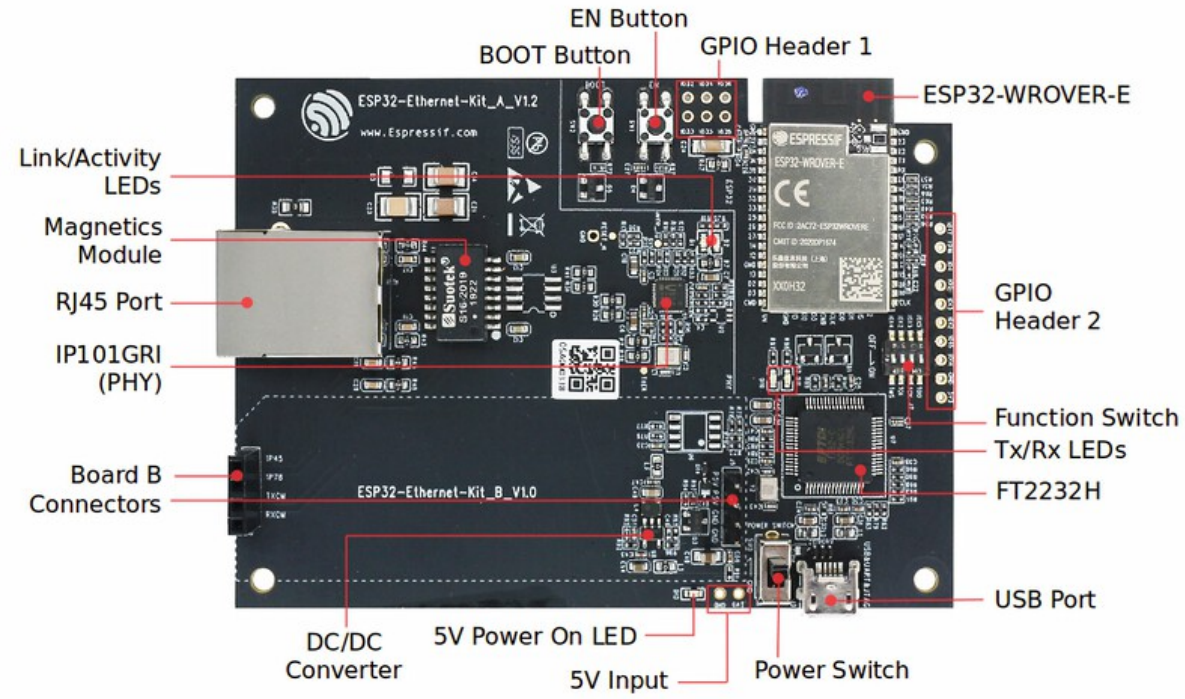


Fig. 23: ESP32-Ethernet-Kit - Ethernet board (A) layout (click to enlarge)

Table 1: Table 1 Component Description

Key Component	Description
ESP32-WROVER-E	This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities.
GPIO Header 2	Five unpopulated through-hole solder pads to provide access to selected GPIOs of ESP32. For details, see *GPIO Header 2*.
Function Switch	A 4-bit DIP switch used to configure the functionality of selected GPIOs of ESP32. For details see *Function Switch*.
Tx/Rx LEDs	Two LEDs to show the status of UART transmission.
FT2232H	The FT2232H chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232H also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232H chip enhances user-friendliness in terms of application development and debugging. See ESP32-Ethernet-Kit V1.2 Ethernet board (A) schematic.
USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power Switch	Power On/Off Switch. Toggling the switch to **5V0** position powers the board on, toggling to **GND** position powers the board off.
5V Input	The 5 V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer).
5V Power On LED	This red LED turns on when power is supplied to the board, either from USB or 5 V Input.
DC/DC Converter	Provided DC 5 V to 3.3 V conversion, output current up to 2 A.
Board B Connectors	A pair male and female header pins for mounting the *PoE board (B)*
IP101GRI (PHY)	The physical layer (PHY) connection to the Ethernet cable is implemented using the IP101GRI chip. The connection between PHY and ESP32 is done through the reduced media-independent interface (RMII), a variant of the media-independent interface (MII) standard. The PHY supports the IEEE 802.3/802.3u standard of 10/100 Mbps.
RJ45 Port	Ethernet network data transmission port.
Magnetics Module	The Magnetics are part of the Ethernet specification to protect against faults and transients, including rejection of common mode signals between the transceiver IC and the cable. The magnetics also provide galvanic isolation between the transceiver and the Ethernet device.
Link/Activity LEDs	Two LEDs (green and red) that respectively indicate the "Link" and "Activity" statuses of the PHY.
BOOT Button	Download button. Holding down **BOOT** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
EN Button	Reset button.
GPIO Header 1	This header provides six unpopulated through-hole solder pads connected to spare GPIOs of ESP32. For details, see *GPIO Header 1*.

Note: Automatic firmware download is supported. If following steps and using software described in Section *Start Application Development*, users don't need to do any operation with BOOT button or EN button.

PoE Board (B) This board coverts power delivered over the Ethernet cable (PoE) to provide a power supply for the Ethernet board (A). The main components of the PoE board (B) are shown on the block diagram under *Functionality Overview*.

The PoE board (B) has the following features:

- Support for IEEE 802.3at
- Power output: 5 V, 1.4 A

To take advantage of the PoE functionality the **RJ45 Port** of the Ethernet board (A) should be connected with an

Ethernet cable to a switch that supports PoE. When the Ethernet board (A) detects 5 V power output from the PoE board (B), the USB power will be automatically cut off.

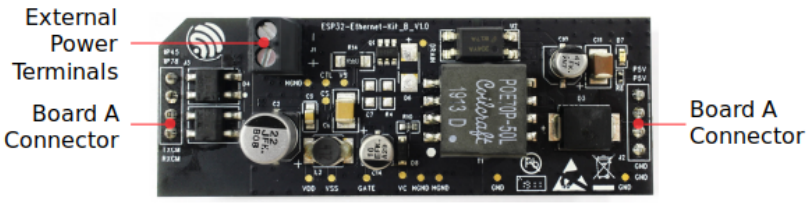


Fig. 24: ESP32-Ethernet-Kit - PoE board (B) layout (click to enlarge)

Table 2: Table PoE board (B)

Key Component	Description
Board A Connector	Four female (left) and four male (right) header pins for connecting the PoE board (B) to *Ethernet board (A)*. The pins on the left accept power coming from a PoE switch. The pins on the right deliver 5 V power supply to the Ethernet board (A).
External Power Terminals	Optional power supply (26.6 ~ 54 V) to the PoE board (B).

Setup Options This section describes options to configure the ESP32-Ethernet-Kit hardware.

Function Switch When in On position, this DIP switch is routing listed GPIOs to FT2232H to provide JTAG functionality. When in Off position, the GPIOs may be used for other purposes.

DIP SW	GPIO Pin
1	GPIO13
2	GPIO12
3	GPIO15
4	GPIO14

RMII Clock Selection The ethernet MAC and PHY under RMII working mode need a common 50 MHz reference clock (i.e. RMII clock) that can be provided either externally, or generated from internal ESP32 APLL (not recommended).

Note: For additional information on the RMII clock selection, please refer to ESP32-Ethernet-Kit V1.2 Ethernet board (A) schematic, sheet 2, location D2.

RMII Clock Sourced Externally by PHY By default, the ESP32-Ethernet-Kit is configured to provide RMII clock for the IP101GRI PHY' s 50M_CLKO output. The clock signal is generated by the frequency multiplication of 25 MHz crystal connected to the PHY. For details, please see the figure below.

Please note that the PHY is reset on power up by pulling the RESET_N signal down with a resistor. ESP32 should assert RESET_N high with GPIO5 to enable PHY. Only this can ensure the power-up of system. Otherwise ESP32 may enter download mode (when the clock signal of REF_CLK_50M is at a high logic level during the GPIO0 power-up sampling phase).

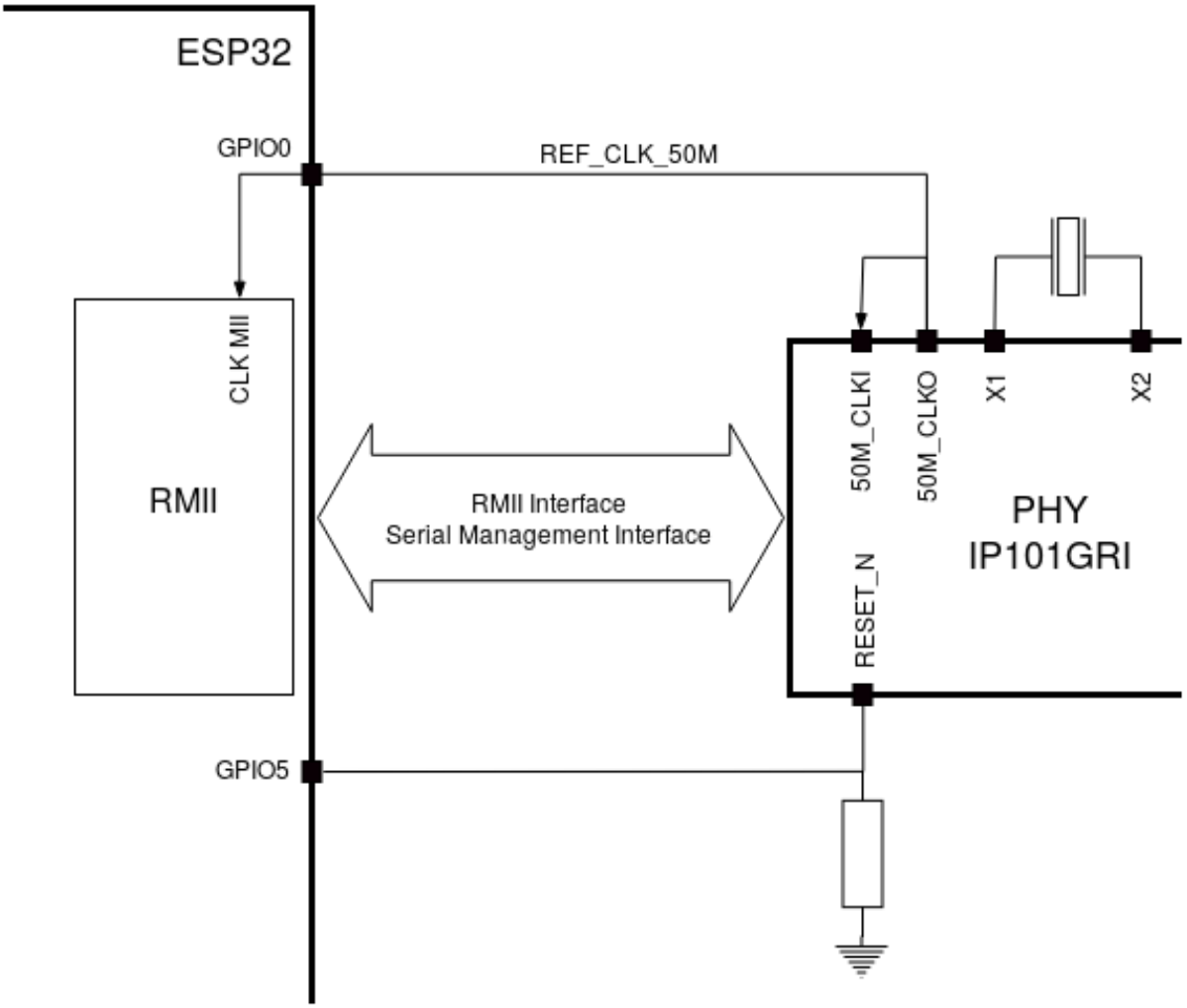


Fig. 25: RMII Clock from IP101GRI PHY

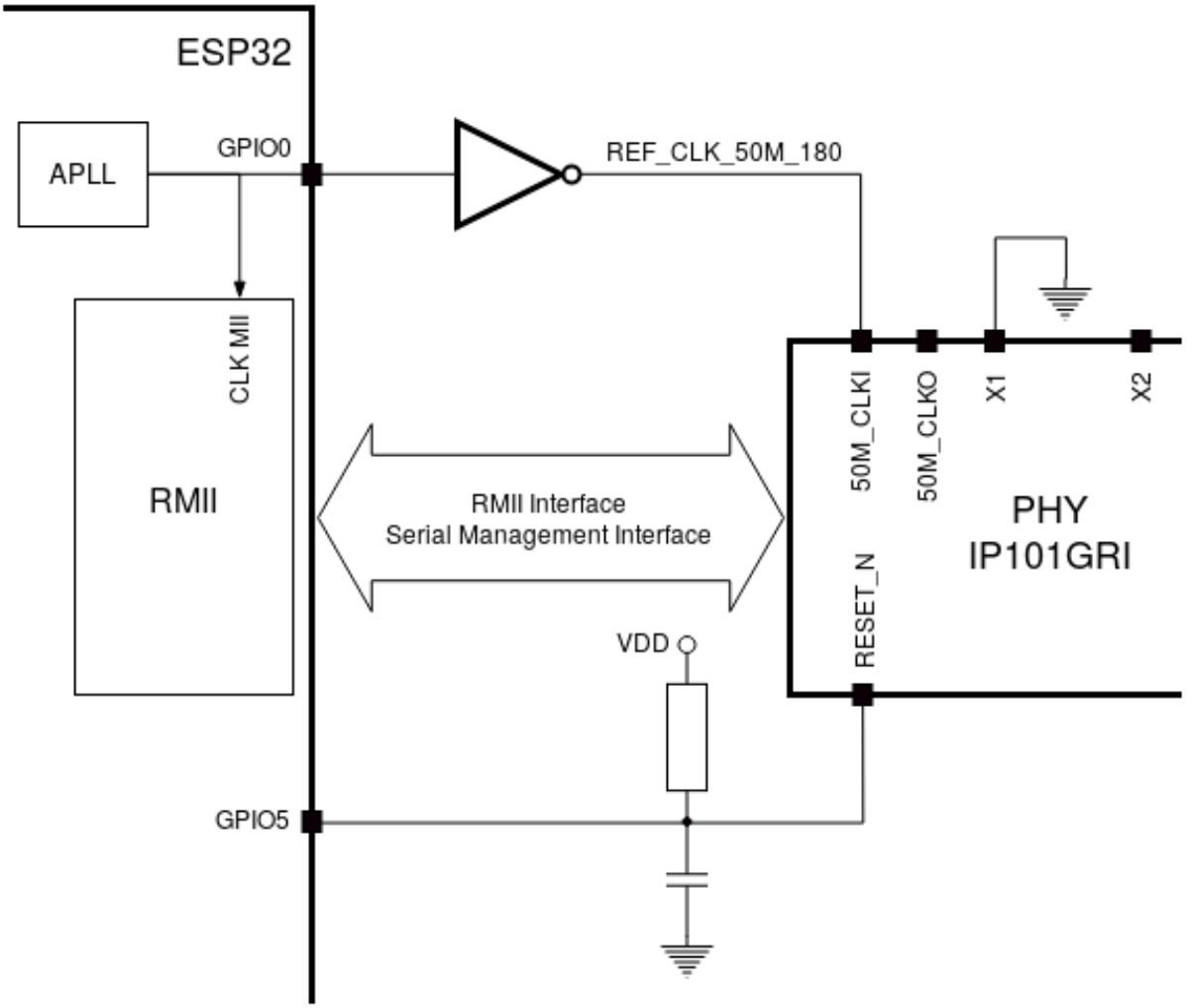


Fig. 26: RMII Clock from ESP Internal APLL

RMII Clock Sourced Internally from ESP32's APLL Another option is to source the RMII Clock from internal ESP32 APLL, see figure below. The clock signal coming from GPIO0 is first inverted, to account for transmission line delay, and then supplied to the PHY.

To implement this option, users need to remove or add some RC components on the board. For details please refer to ESP32-Ethernet-Kit V1.2 Ethernet board (A) schematic, sheet 2, location D2. Please note that if the APLL is already used for other purposes (e.g. I2S peripheral), then you have no choice but use an external RMII clock.

GPIO Allocation This section describes allocation of ESP32 GPIOs to specific interfaces or functions of the ESP32-Ethernet-Kit.

IP101GRI (PHY) Interface The allocation of the ESP32 (MAC) pins to IP101GRI (PHY) is shown in the table below. Implementation of ESP32-Ethernet-Kit defaults to Reduced Media-Independent Interface (RMII).

No.	ESP32 Pin (MAC)	IP101GRI (PHY)
RMII Interface		
1	GPIO21	TX_EN
2	GPIO19	TXD[0]
3	GPIO22	TXD[1]
4	GPIO25	RXD[0]
5	GPIO26	RXD[1]
6	GPIO27	CRS_DV
7	GPIO0	REF_CLK
Serial Management Interface		
8	GPIO23	MDC
9	GPIO18	MDIO
PHY Reset		
10	GPIO5	Reset_N

Note: The allocation of all pins under the ESP32's *RMII Interface* is fixed and cannot be changed either through IO MUX or GPIO Matrix. REF_CLK can only be selected from GPIO0, GPIO16 or GPIO17 and it can not be changed through GPIO Matrix.

GPIO Header 1 This header exposes some GPIOs that are not used elsewhere on the ESP32-Ethernet-Kit.

No.	ESP32 Pin
1	GPIO32
2	GPIO33
3	GPIO34
4	GPIO35
5	GPIO36
6	GPIO39

GPIO Header 2 This header contains GPIOs that may be used for other purposes depending on scenarios described in column "Comments".

No.	ESP32 Pin	Comments
1	GPIO17	See note 1
2	GPIO16	See note 1
3	GPIO4	
4	GPIO2	
5	GPIO13	See note 2
6	GPIO12	See note 2
7	GPIO15	See note 2
8	GPIO14	See note 2
9	GND	Ground
10	3V3	3.3 V power supply

Note:

1. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-E module and therefore not available for use. If you need to use these pins, please solder a module without PSRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.
2. Functionality depends on the settings of the *Function Switch*.

GPIO Allocation Summary

ESP32-WROVER-E	IP101GRI	UART	JTAG	GPIO	Comments
S_VP				IO36	
S_VN				IO39	
IO34				IO34	
IO35				IO35	
IO32				IO32	
IO33				IO33	
IO25	RXD[0]				
IO26	RXD[1]				
IO27	CRS_DV				
IO14			TMS	IO14	
IO12			TDI	IO12	
IO13			TCK	IO13	
IO15			TDO	IO15	
IO2				IO2	
IO0	REF_CLK				See note 1
IO4				IO4	
IO16				IO16 (NC)	See note 2
IO17				IO17 (NC)	See note 2
IO5	Reset_N				See note 1
IO18	MDIO				
IO19	TXD[0]				
IO21	TX_EN				
RXD0		RXD			
TXD0		TXD			
IO22	TXD[1]				
IO23	MDC				

Note:

1. To prevent the power-on state of the GPIO0 from being affected by the clock output on the PHY side, the RESET_N signal to PHY defaults to low, turning the clock output off. After power-on you can control RE-

SET_N with GPIO5 to turn the clock output on. See also *RMII Clock Sourced Externally by PHY*. For PHYs that cannot turn off the clock output through RESET_N, it is recommended to use a crystal module that can be disabled/enabled externally. Similarly like when using RESET_N, the oscillator module should be disabled by default and turned on by ESP32 after power-up. For a reference design please see ESP32-Ethernet-Kit V1.2 Ethernet board (A) schematic.

2. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-E module and therefore not available for use. If you need to use these pins, please solder a module without PSRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.

Start Application Development Before powering up your ESP32-Ethernet-Kit, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup

1. Set the **Function Switch** on the *Ethernet board (A)* to its default position by turning all the switches to **ON**.
2. To simplify flashing and testing of the application, do not input extra signals to the board headers.
3. The *PoE board (B)* can now be plugged in, but do not connect external power to it.
4. Connect the *Ethernet board (A)* to the PC with a USB cable.
5. Turn the **Power Switch** from GND to 5V0 position, the **5V Power On LED** should light up.

Now to Development Proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Move on to the next section only if you have successfully completed all the above steps.

Configure and Load the Ethernet Example After setting up the development environment and testing the board, you can configure and flash the ethernet/basic example. This example has been created for testing Ethernet functionality. It supports different PHY, including **IP101GRI** installed on *ESP32-Ethernet-Kit V1.2 (click to enlarge)*.

Summary of Changes from ESP32-Ethernet-Kit V1.1

- Correct the placement of GPIO pin number marking on the board’ s silkscreen besides the DIP switch.
- Values of C1, C2, C42, and C43 are updated to 20 pF. For more information, please check ESP32-Ethernet-Kit V1.2 Ethernet board (A) schematic.
- Replace ESP32-WROVER-B with ESP32-WROVER-E.

Other Versions of ESP32-Ethernet-Kit

- *ESP32-Ethernet-Kit V1.0 Getting Started Guide*
- *ESP32-Ethernet-Kit V1.1 Getting Started Guide*

Related Documents

- ESP32-Ethernet-Kit V1.2 Ethernet Board (A) Schematic (PDF)
- ESP32-Ethernet-Kit PoE Board (B) Schematic (PDF)
- ESP32-Ethernet-Kit V1.2 Ethernet Board (A) PCB Layout (PDF)
- ESP32-Ethernet-Kit PoE Board (B) PCB Layout (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER-E Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

For other design documentation for the board, please contact us at sales@espressif.com.

ESP32-Ethernet-Kit V1.0 Getting Started Guide

This guide shows how to get started with the ESP32-Ethernet-Kit development board and also provides information about its functionality and configuration options.

The *ESP32-Ethernet-Kit* is an Ethernet-to-Wi-Fi development board that enables Ethernet devices to be interconnected over Wi-Fi. At the same time, to provide more flexible power supply options, the ESP32-Ethernet-Kit also supports power over Ethernet (PoE).

What You Need

- *ESP32-Ethernet-Kit V1.0 board*
- USB 2.0 A to Micro B Cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-Ethernet-Kit is an ESP32-based development board produced by Espressif.

It consists of two development boards, the Ethernet board A and the PoE board B. The *Ethernet board (A)* contains Bluetooth / Wi-Fi dual-mode ESP32-WROVER-B module and IP101GRI, a Single Port 10/100 Fast Ethernet Transceiver (PHY). The *PoE board (B)* provides power over Ethernet functionality. The A board can work independently, without the board B installed.

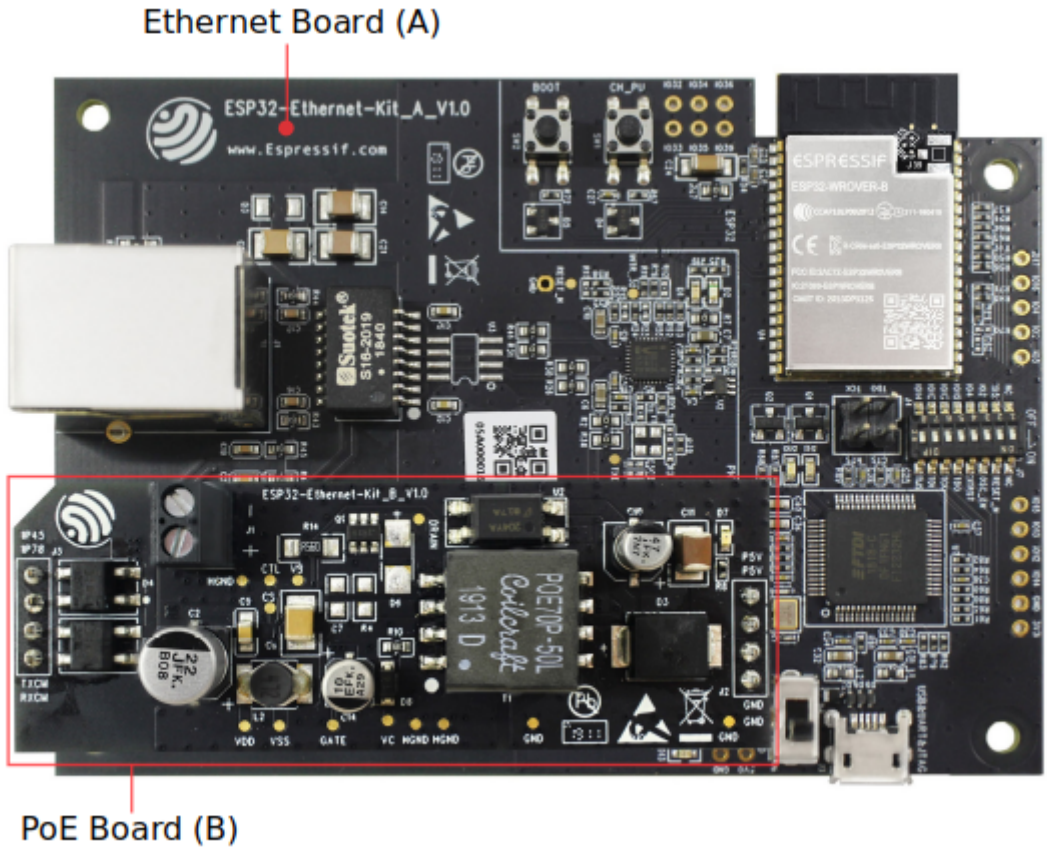


Fig. 27: ESP32-Ethernet-Kit V1.0

For the application loading and monitoring the Ethernet board (A) also features FTDI FT2232H chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface

without a separate JTAG debugger.

Functionality Overview The block diagram below shows the main components of ESP32-Ethernet-Kit and their interconnections.

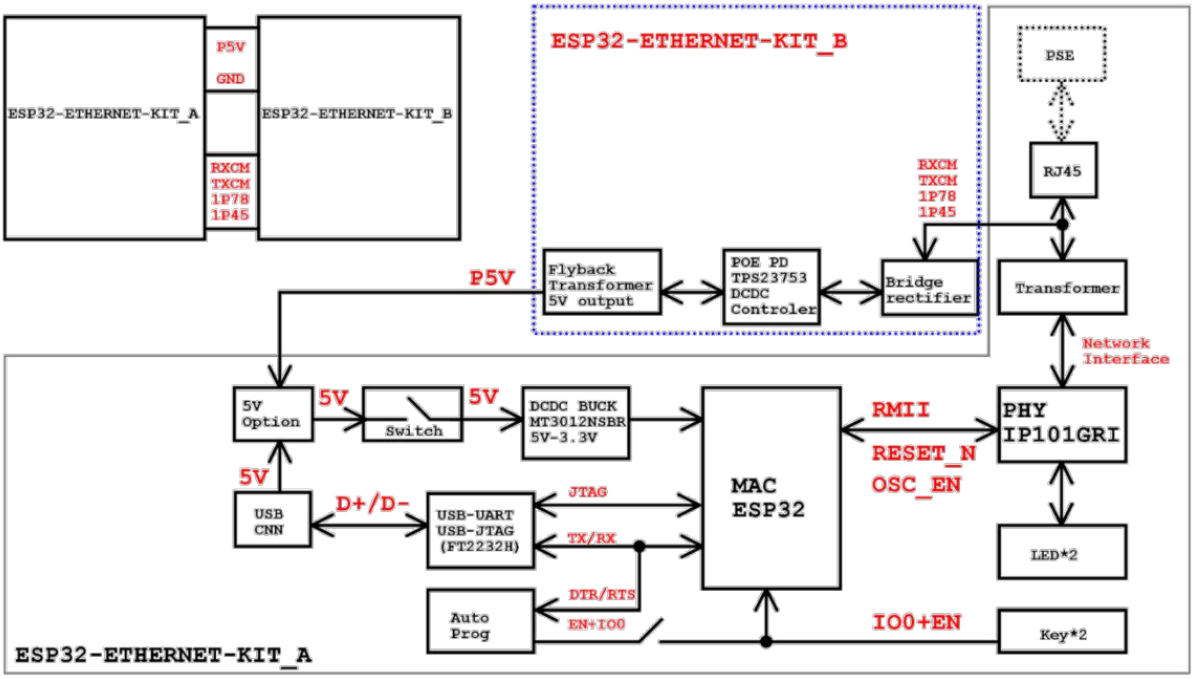


Fig. 28: ESP32-Ethernet-Kit block diagram (click to enlarge)

Functional Description The following two figures and tables describe the key components, interfaces, and controls of the ESP32-Ethernet-Kit.

Ethernet Board (A) The table below provides description starting from the picture's top right corner and going clockwise.

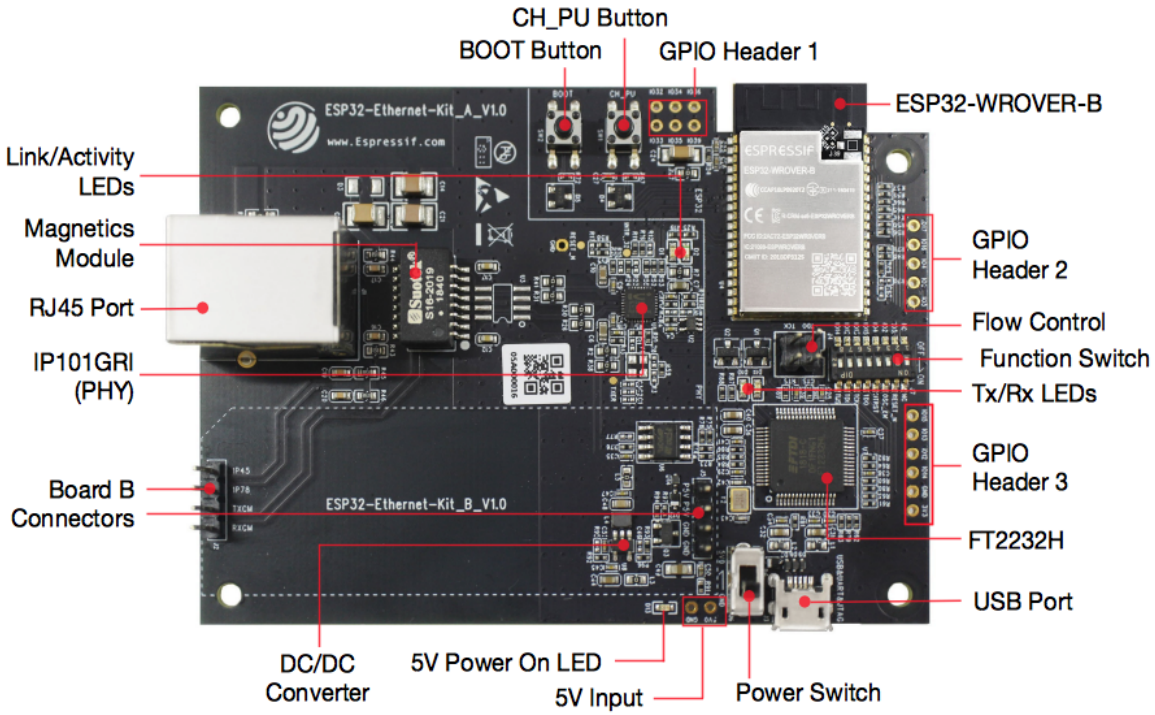


Fig. 29: ESP32-Ethernet-Kit - Ethernet board (A) layout (click to enlarge)

Key Component	Description
ESP32-WROVER-B	This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities.
GPIO Header 2	Five unpopulated through-hole solder pads to provide access to selected GPIOs of ESP32. For details, see *GPIO Header 2*.
Flow Control	A jumper header with access to the board signals. For details, see *Flow Control*.
Function Switch	A DIP switch used to configure the functionality of selected GPIOs of ESP32. For details, see *Function Switch*.
Tx/Rx LEDs	Two LEDs to show the status of UART transmission.
GPIO Header 3	Provides access to some GPIOs of ESP32 that can be used depending on the position of the *Function Switch*.
FT2232H	The FT2232H chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232H also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232H chip enhances user-friendliness in terms of application development and debugging. See ESP32-Ethernet-Kit V1.0 Ethernet board (A) schematic.
USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power Switch	Power On/Off Switch. Toggling toward the **Boot** button powers the board on, toggling away from **Boot** powers the board off.
5V Input	The 5V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer).
5V Power On LED	This red LED turns on when power is supplied to the board, either from USB or 5V Input.

PoE Board (B) This board coverts power delivered over the Ethernet cable (PoE) to provide a power supply for the Ethernet board (A). The main components of the PoE board (B) are shown on the block diagram under *Functionality Overview*.

The PoE board (B) has the following features:

- Support for IEEE 802.3at
- Power output: 5 V, 1.4 A

To take advantage of the PoE functionality the **RJ45 Port** of the Ethernet board (A) should be connected with an Ethernet cable to a switch that supports PoE. When the Ethernet board (A) detects 5 V power output from the PoE board (B), the USB power will be automatically cut off.

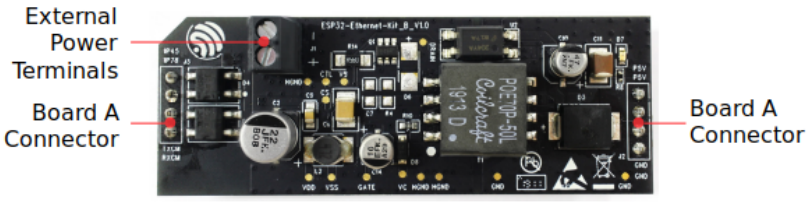


Fig. 30: ESP32-Ethernet-Kit - PoE board (B) layout (click to enlarge)

Key Component	Description
Board A Connector	Four female header pins for mounting this board onto *Ethernet board (A)*.
External Power Terminals	Optional power supply to the PoE board (B).

Setup Options This section describes options to configure the ESP32-Ethernet-Kit hardware.

Function Switch The functions for specific GPIO pins can be selected with the **Function Switch**.

DIP SW	GPIO Pin	Pin Functionality if DIP SW is ON
1	GPIO14	Connected to FT2232H to provide JTAG functionality
2	GPIO12	Connected to FT2232H to provide JTAG functionality
3	GPIO13	Connected to FT2232H to provide JTAG functionality
4	GPIO15	Connected to FT2232H to provide JTAG functionality
5	GPIO4	Connected to FT2232H to provide JTAG functionality
6	GPIO2	Connected to on-board 25 MHz oscillator
7	GPIO5	Connected to RESET_N input of IP101GRI
8	n/a	

You can make a certain GPIO pin available for other purposes by putting its DIP SW to the Off position.

Flow Control This is a 2 x 2 jumper pin header intended for the UART flow control.

.	Signal	Comment
1	MTDO	GPIO13, see also *Function Switch*
2	MTCK	GPIO15, see also *Function Switch*
3	RTS	RTS signal of FT2232H
4	CTS	CTS signal of FT2232H

GPIO Allocation This section describes allocation of ESP32 GPIOs to specific interfaces or functions of the ESP32-Ethernet-Kit.

IP101GRI (PHY) Interface The allocation of the ESP32 (MAC) pins to IP101GRI (PHY) is shown in the table below. Implementation of ESP32-Ethernet-Kit defaults to Reduced Media-Independent Interface (RMII).

.	ESP32 Pin (MAC)	IP101GRI (PHY)
RMII Interface		
1	GPIO21	TX_EN
2	GPIO19	TXD[0]
3	GPIO22	TXD[1]
4	GPIO25	RXD[0]
5	GPIO26	RXD[1]
6	GPIO27	CRS_DV
7	GPIO0	REF_CLK
Serial Management Interface		
8	GPIO23	MDC
9	GPIO18	MDIO
PHY Reset		
10	GPIO5	Reset_N

Note: Except for REF_CLK, the allocation of all pins under the *RMII Interface* is fixed and cannot be changed either through IOMUX or GPIO Matrix.

GPIO Header 1 This header exposes some GPIOs that are not used elsewhere on the ESP32-Ethernet-Kit.

.	ESP32 Pin
1	GPIO32
2	GPIO33
3	GPIO34
4	GPIO35
5	GPIO36
6	GPIO39

GPIO Header 2 This header contains the GPIOs with specific MII functionality (except GPIO2), as opposed to Reduced Media-Independent Interface (RMII) functionality implemented on ESP32-Ethernet-Kit board by default, see *IP101GRI (PHY) Interface*. Depending on the situation, if MMI is used, specific Ethernet applications might require this functionality.

.	ESP32 Pin	MII Function	Comments
1	GPIO17	EMAC_CLK_180	See note 1
2	GPIO16	EMAC_CLK_OUT	See note 1
3	GPIO4	EMAC_TX_ER	
4	GPIO2	n/a	See note 2
5	GPIO5	EMAC_RX_CLK	See note 2

Note:

1. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-B module and therefore not available for use. If you need to use these pins, please solder a module without SPIRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.
2. Functionality depends on the settings of the *Function Switch*.

GPIO Header 3 The functionality of GPIOs connected to this header depends on the settings of the *Function Switch*.

.	ESP32 Pin
1	GPIO15
2	GPIO13
3	GPIO12
4	GPIO14
5	GND
6	3V3

GPIO Allocation Summary

ESP32-WROVER-B	IP101GRI	UART	JTAG	GPIO	Comments
S_VP				IO36	
S_VN				IO39	
IO34				IO34	
IO35				IO35	
IO32				IO32	
IO33				IO33	
IO25	RXD[0]				
IO26	RXD[1]				
IO27	CRS_DV				
IO14			TMS	IO14	
IO12			TDI	IO12	
IO13		RTS	TCK	IO13	
IO15		CTS	TDO	IO15	
IO2				IO2	See notes 1 and 3 below
IO0	REF_CLK				See notes 2 and 3 below
IO4			nTRST	IO4	
IO16				IO16 (NC)	See note 4 below
IO17				IO17 (NC)	See note 4 below
IO5	Reset_N			IO5	
IO18	MDIO				
IO19	TXD[0]				
IO21	TX_EN				
RXD0		RXD			
TXD0		TXD			
IO22	TXD[1]				
IO23	MDC				

Note:

1. GPIO2 is used to enable external oscillator of the PHY.
2. GPIO0 is a source of 50 MHz reference clock for the PHY. The clock signal is first inverted, to account for transmission line delay, and then supplied to the PHY.
3. To prevent affecting the power-on state of GPIO0 by the clock output on the PHY side, the PHY external oscillator is enabled using GPIO2 after ESP32 is powered up.
4. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-B module and therefore not available for use. If you need to use these pins, please solder a module without SPIRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.

Start Application Development Before powering up your ESP32-Ethernet-Kit, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup

1. Set the **Function Switch** on the *Ethernet board (A)* to its default position by turning all the switches to **ON**.
2. To simplify flashing and testing the application, do not install any jumpers and do not connect any signals to the board headers.
3. The *PoE board (B)* can now be plugged in, but do not connect external power to it.
4. Connect the *Ethernet board (A)* to the PC with a USB cable.
5. Turn the **Power Switch** from GND to 5V0 position, the **5V Power On LED** should light up.

Now to Development Proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

To use the older GNU Make compilation system, please refer to *Installation* section.

Move on to the next section only if you have successfully completed all the above steps.

Configure and Load the Ethernet Example After setting up the development environment and testing the board, you can configure and flash the ethernet/basic example. This example has been created for testing Ethernet functionality. It supports different PHY, including **IP101GRI** installed on *ESP32-Ethernet-Kit V1.0 board*.

Related Documents

- ESP32-Ethernet-Kit V1.0 Ethernet board (A) schematic (PDF)
- ESP32-Ethernet-Kit V1.0 PoE board (B) schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER-B Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

For other design documentation for the board, please contact us at sales@espressif.com.

ESP32-Ethernet-Kit V1.1 Getting Started Guide

This guide shows how to get started with the ESP32-Ethernet-Kit development board and also provides information about its functionality and configuration options.

The *ESP32-Ethernet-Kit* is an Ethernet-to-Wi-Fi development board that enables Ethernet devices to be interconnected over Wi-Fi. At the same time, to provide more flexible power supply options, the ESP32-Ethernet-Kit also supports power over Ethernet (PoE).

What You Need

- *ESP32-Ethernet-Kit V1.1 board*
- USB 2.0 A to Micro B Cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section *Start Application Development*.

Overview ESP32-Ethernet-Kit is an ESP32-based development board produced by Espressif.

It consists of two development boards, the Ethernet board A and the PoE board B. The *Ethernet board (A)* contains Bluetooth / Wi-Fi dual-mode ESP32-WROVER-B module and IP101GRI, a Single Port 10/100 Fast Ethernet Transceiver (PHY). The *PoE board (B)* provides power over Ethernet functionality. The A board can work independently, without the board B installed.

For the application loading and monitoring, the Ethernet board (A) also features FTDI FT2232H chip - an advanced multi-interface USB bridge. This chip enables to use JTAG for direct debugging of ESP32 through the USB interface without a separate JTAG debugger.

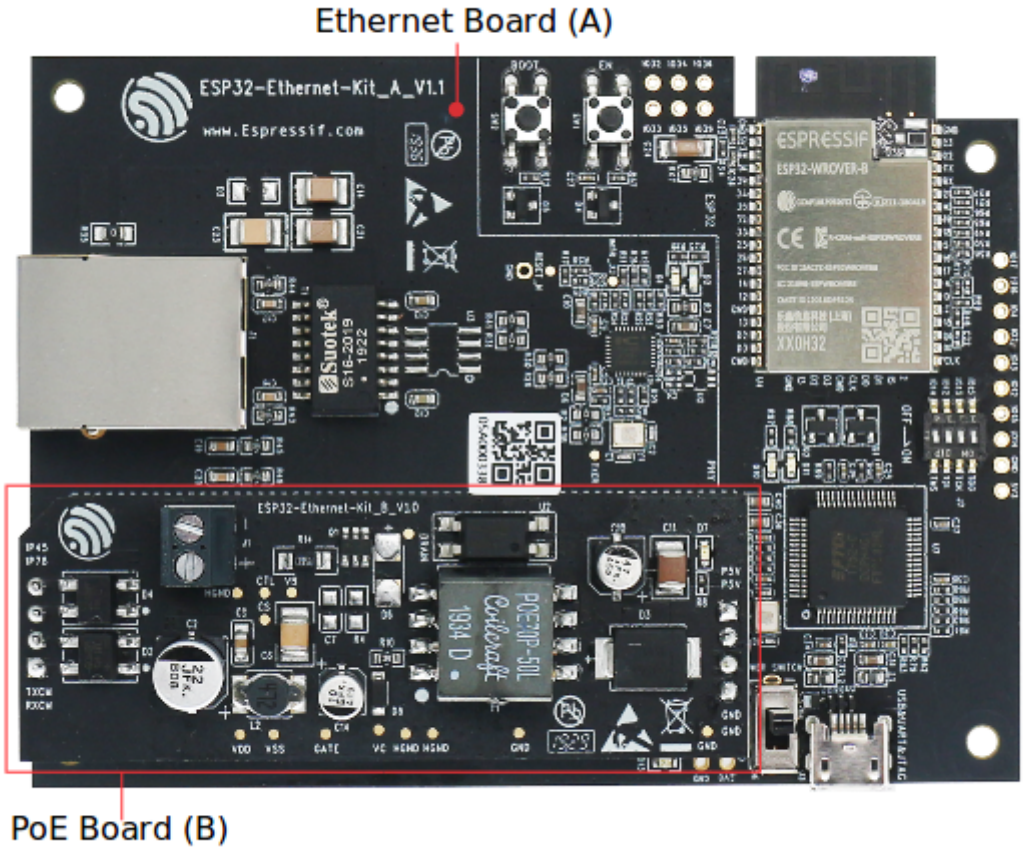


Fig. 31: ESP32-Ethernet-Kit V1.1

Functionality Overview The block diagram below shows the main components of ESP32-Ethernet-Kit and their interconnections.

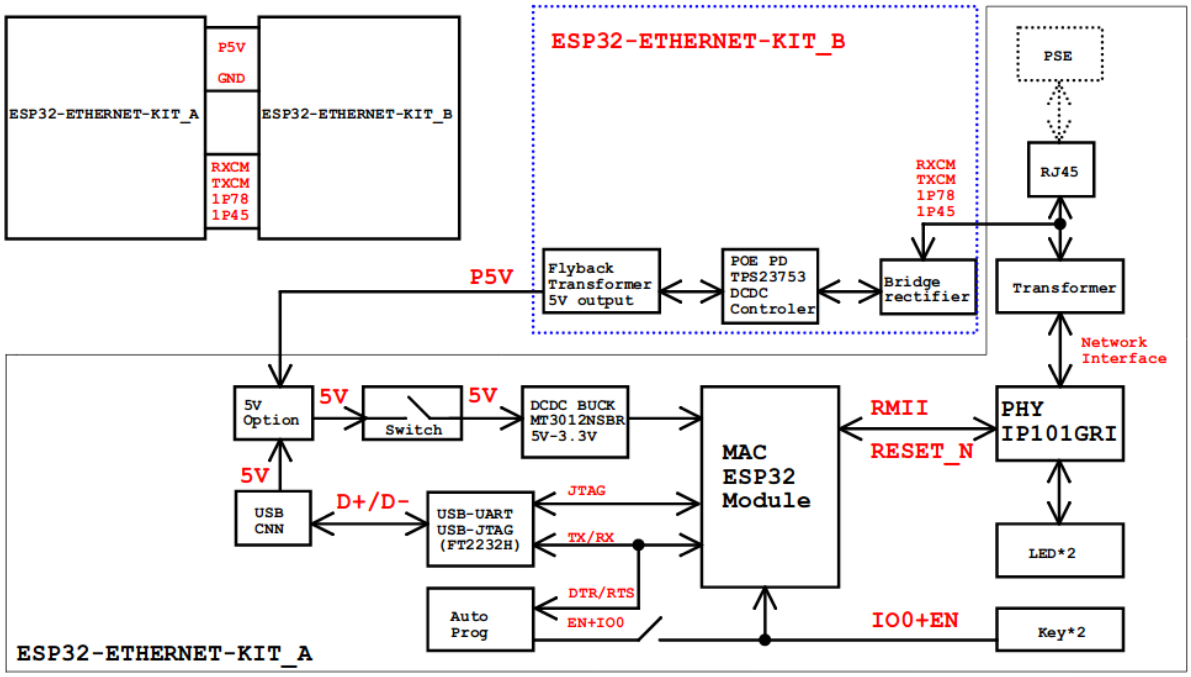


Fig. 32: ESP32-Ethernet-Kit block diagram (click to enlarge)

Functional Description The following figures and tables describe the key components, interfaces, and controls of the ESP32-Ethernet-Kit.

Ethernet Board (A) The table below provides description starting from the picture's top right corner and going clockwise.

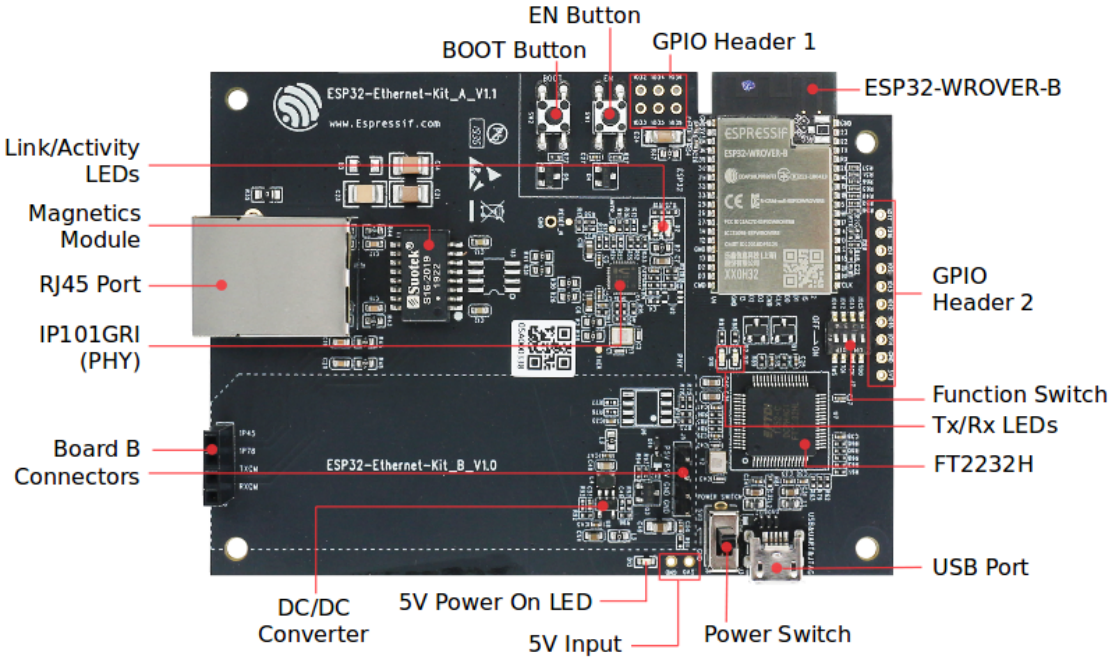


Fig. 33: ESP32-Ethernet-Kit - Ethernet board (A) layout (click to enlarge)

Table 3: Table 1 Component Description

Key Component	Description
ESP32-WROVER-B	This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities.
GPIO Header 2	Five unpopulated through-hole solder pads to provide access to selected GPIOs of ESP32. For details, see *GPIO Header 2*.
Function Switch	A 4-bit DIP switch used to configure the functionality of selected GPIOs of ESP32. Please note that placement of GPIO pin number marking on the board' s silkscreen besides the DIP switch is incorrect. For details and correct pin allocation see *Function Switch*.
Tx/Rx LEDs	Two LEDs to show the status of UART transmission.
FT2232H	The FT2232H chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232H also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232H chip enhances user-friendliness in terms of application development and debugging. See ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic.
USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
Power Switch	Power On/Off Switch. Toggling the switch to **5V0** position powers the board on, toggling to **GND** position powers the board off.
5V Input	The 5 V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer).
5V Power On LED	This red LED turns on when power is supplied to the board, either from USB or 5 V Input.
DC/DC Converter	Provided DC 5 V to 3.3 V conversion, output current up to 2 A.
Board B Connectors	A pair male and female header pins for mounting the *PoE board (B)*.
IP101GRI (PHY)	The physical layer (PHY) connection to the Ethernet cable is implemented using the IP101GRI chip. The connection between PHY and ESP32 is done through the reduced media-independent interface (RMII), a variant of the media-independent interface (MII) standard. The PHY supports the IEEE 802.3 / 802.3u standard of 10/100 Mbps.
RJ45 Port	Ethernet network data transmission port.
Magnetics Module	The Magnetics are part of the Ethernet specification to protect against faults and transients, including rejection of common mode signals between the transceiver IC and the cable. The magnetics also provide galvanic isolation between the transceiver and the Ethernet device.

PoE Board (B) This board coverts power delivered over the Ethernet cable (PoE) to provide a power supply for the Ethernet board (A). The main components of the PoE board (B) are shown on the block diagram under *Functionality Overview*.

The PoE board (B) has the following features:

- Support for IEEE 802.3at
- Power output: 5 V, 1.4 A

To take advantage of the PoE functionality the **RJ45 Port** of the Ethernet board (A) should be connected with an Ethernet cable to a switch that supports PoE. When the Ethernet board (A) detects 5 V power output from the PoE board (B), the USB power will be automatically cut off.

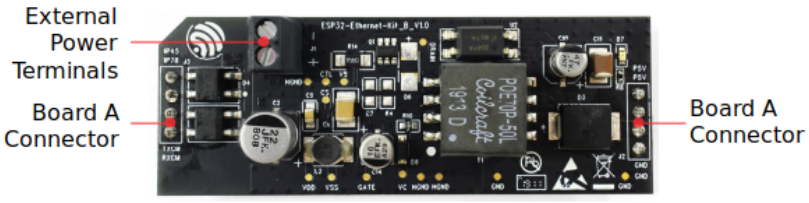


Fig. 34: ESP32-Ethernet-Kit - PoE board (B) layout (click to enlarge)

Table 4: Table PoE board (B)

Key Component	Description
Board A Connector	Four female (left) and four male (right) header pins for connecting the PoE board (B) to *Ethernet board (A)*. The pins on the left accept power coming from a PoE switch. The pins on the right deliver 5 V power supply to the Ethernet board (A).
External Power Terminals	Optional power supply (26.6 ~ 54 V) to the PoE board (B).

Setup Options This section describes options to configure the ESP32-Ethernet-Kit hardware.

Function Switch When in On position, this DIP switch is routing listed GPIOs to FT2232H to provide JTAG functionality. When in Off position, the GPIOs may be used for other purposes.

DIP SW	GPIO Pin
1	GPIO13
2	GPIO12
3	GPIO15
4	GPIO14

Note: Placement of GPIO pin number marking on the board' s silkscreen besides the DIP switch is incorrect. Please use instead the pin order as in the table above.

RMII Clock Selection The ethernet MAC and PHY under RMII working mode need a common 50 MHz reference clock (i.e. RMII clock) that can be provided either externally, or generated from internal ESP32 APLL.

Note: For additional information on the RMII clock selection, please refer to ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic, sheet 2, location D2.

RMII Clock Sourced Externally by PHY By default, the ESP32-Ethernet-Kit is configured to provide RMII clock for the IP101GRI PHY's 50M_CLKO output. The clock signal is generated by the frequency multiplication of 25 MHz crystal connected to the PHY. For details, please see the figure below.

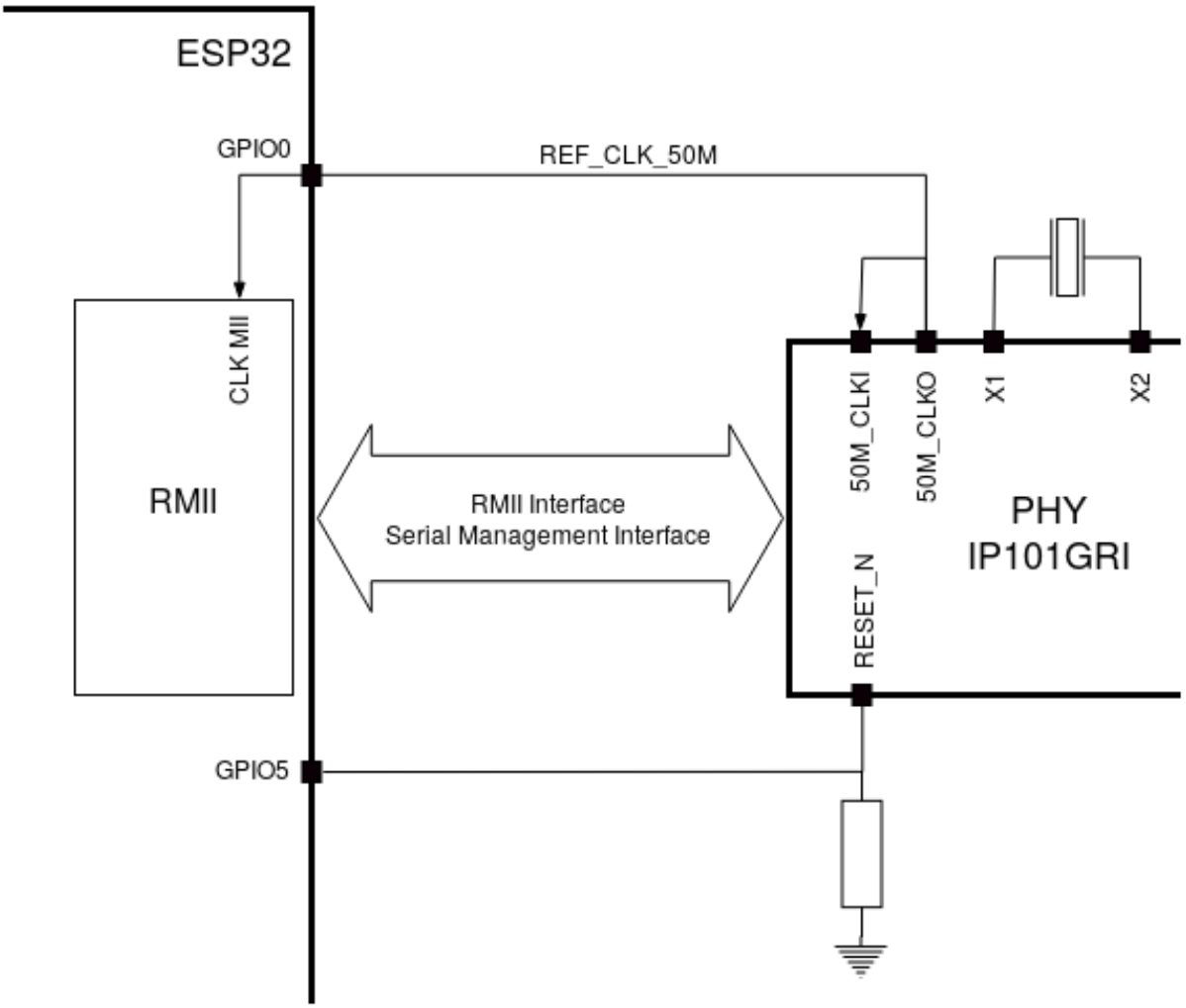


Fig. 35: RMII Clock from IP101GRI PHY

Please note that the PHY is reset on power up by pulling the RESET_N signal down with a resistor. ESP32 should assert RESET_N high with GPIO5 to enable PHY. Only this can ensure the power-up of system. Otherwise ESP32 may enter download mode (when the clock signal of REF_CLK_50M is at a high logic level during the GPIO0 power-up sampling phase).

RMII Clock Sourced Internally from ESP32's APLL Another option is to source the RMII Clock from internal ESP32 APLL, see figure below. The clock signal coming from GPIO0 is first inverted, to account for transmission line delay, and then supplied to the PHY.

To implement this option, users need to remove or add some RC components on the board. For details please refer to ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic, sheet 2, location D2. Please note that if the APLL is already used for other purposes (e.g. I2S peripheral), then you have no choice but use an external RMII clock.

GPIO Allocation This section describes allocation of ESP32 GPIOs to specific interfaces or functions of the ESP32-Ethernet-Kit.

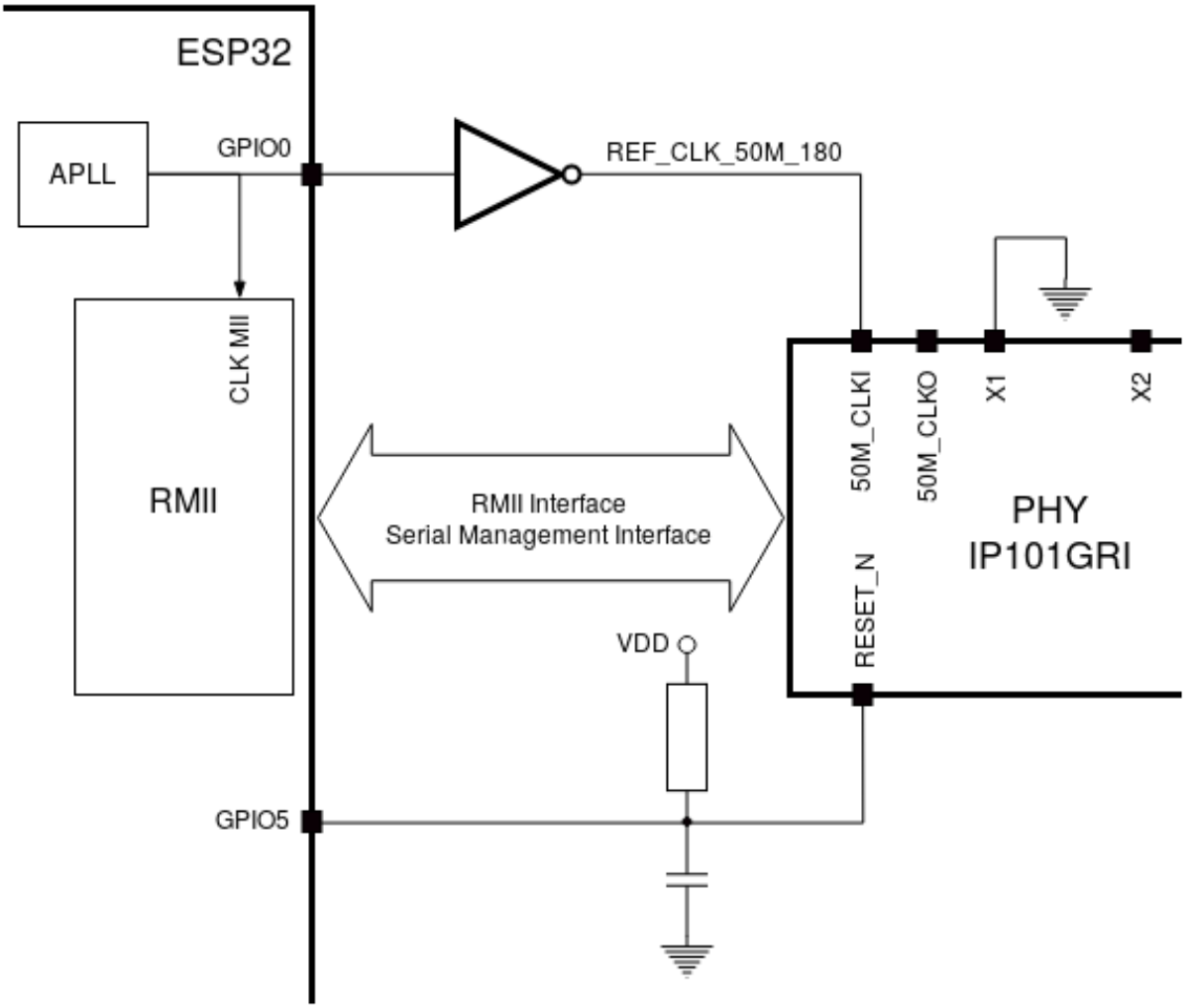


Fig. 36: RMII Clock from ESP Internal APLL

IP101GRI (PHY) Interface The allocation of the ESP32 (MAC) pins to IP101GRI (PHY) is shown in the table below. Implementation of ESP32-Ethernet-Kit defaults to Reduced Media-Independent Interface (RMII).

.	ESP32 Pin (MAC)	IP101GRI (PHY)
RMII Interface		
1	GPIO21	TX_EN
2	GPIO19	TXD[0]
3	GPIO22	TXD[1]
4	GPIO25	RXD[0]
5	GPIO26	RXD[1]
6	GPIO27	CRS_DV
7	GPIO0	REF_CLK
Serial Management Interface		
8	GPIO23	MDC
9	GPIO18	MDIO
PHY Reset		
10	GPIO5	Reset_N

Note: Except for REF_CLK, the allocation of all pins under the ESP32' s *RMII Interface* is fixed and cannot be changed either through IOMUX or GPIO Matrix.

GPIO Header 1 This header exposes some GPIOs that are not used elsewhere on the ESP32-Ethernet-Kit.

.	ESP32 Pin
1	GPIO32
2	GPIO33
3	GPIO34
4	GPIO35
5	GPIO36
6	GPIO39

GPIO Header 2 This header contains GPIOs that may be used for other purposes depending on scenarios described in column "Comments".

.	ESP32 Pin	Comments
1	GPIO17	See note 1
2	GPIO16	See note 1
3	GPIO4	
4	GPIO2	
5	GPIO13	See note 2
6	GPIO12	See note 2
7	GPIO15	See note 2
8	GPIO14	See note 2
9	GND	Ground
10	3V3	3.3 V power supply

Note:

1. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-B module and therefore not available for use. If you need to use these pins, please solder a module without PSRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.
2. Functionality depends on the settings of the *Function Switch*.

GPIO Allocation Summary

ESP32-WROVER-B	IP101GRI	UART	JTAG	GPIO	Comments
S_VP				IO36	
S_VN				IO39	
IO34				IO34	
IO35				IO35	
IO32				IO32	
IO33				IO33	
IO25	RXD[0]				
IO26	RXD[1]				
IO27	CRS_DV				
IO14			TMS	IO14	
IO12			TDI	IO12	
IO13		RTS	TCK	IO13	
IO15		CTS	TDO	IO15	
IO2				IO2	
IO0	REF_CLK				See note 1
IO4				IO4	
IO16				IO16 (NC)	See note 2
IO17				IO17 (NC)	See note 2
IO5	Reset_N				See note 1
IO18	MDIO				
IO19	TXD[0]				
IO21	TX_EN				
RXD0		RXD			
TXD0		TXD			
IO22	TXD[1]				
IO23	MDC				

Note:

1. To prevent the power-on state of the GPIO0 from being affected by the clock output on the PHY side, the RESET_N signal to PHY defaults to low, turning the clock output off. After power-on you can control RESET_N with GPIO5 to turn the clock output on. See also *RMII Clock Sourced Externally by PHY*. For PHYs that cannot turn off the clock output through RESET_N, it is recommended to use a crystal module that can be disabled / enabled externally. Similarly like when using RESET_N, the oscillator module should be disabled by default and turned on by ESP32 after power-up. For a reference design please see ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic.
2. The ESP32 pins GPIO16 and GPIO17 are not broken out to the ESP32-WROVER-B module and therefore not available for use. If you need to use these pins, please solder a module without PSRAM memory inside, e.g. the ESP32-WROOM-32D or ESP32-SOLO-1.

Start Application Development Before powering up your ESP32-Ethernet-Kit, please make sure that the board is in good condition with no obvious signs of damage.

Initial Setup

1. Set the **Function Switch** on the *Ethernet board (A)* to its default position by turning all the switches to **ON**.
2. To simplify flashing and testing of the application, do not input extra signals to the board headers.
3. The *PoE board (B)* can now be plugged in, but do not connect external power to it.
4. Connect the *Ethernet board (A)* to the PC with a USB cable.

5. Turn the **Power Switch** from GND to 5V0 position, the **5V Power On LED** should light up.

Now to Development Proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an example project onto your board.

Move on to the next section only if you have successfully completed all the above steps.

Configure and Load the Ethernet Example After setting up the development environment and testing the board, you can configure and flash the ethernet/basic example. This example has been created for testing Ethernet functionality. It supports different PHY, including **IP101GRI** installed on *ESP32-Ethernet-Kit V1.1*.

Summary of Changes from ESP32-Ethernet-Kit V1.0

- The original inverted clock provided to the PHY by ESP32 using GPIO0 has been replaced by a clock generated on PHY side. The PHY's clock is connected to the ESP32 with same GPIO0. The GPIO2 which was originally used to control the active crystal oscillator on the PHY side, can now be used for other purposes.
- On power up, the ESP32 boot strapping pin GPIO0 may be affected by clock generated on the PHY side. To resolve this issue the PHY's Reset-N signal is pulled low using resistor R17 and effectively turning off the PHY's clock output. The Reset-N signal can be then pulled high by ESP32 using GPIO5.
- Removed FT2232H chip's external SPI Flash U6.
- Removed flow control jumper header J4.
- Removed nTRST JTAG signal. The corresponding GPIO4 can now be used for other purposes.
- Pull-up resistor R68 on the GPIO15 line is moved to the MTDO side of JTAG.
- To make the A and B board connections more foolproof (reduce chances of plugging in the B board in reverse orientation), the original two 4-pin male rows on board A were changed to one 4-pin female row and one 4-pin male row. Corresponding male and female 4-pins rows were installed on board B.

Other Versions of ESP32-Ethernet-Kit

- *ESP32-Ethernet-Kit V1.0 Getting Started Guide*

Related Documents

- ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic (PDF)
- ESP32-Ethernet-Kit V1.0 PoE board (B) schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER-B Datasheet (PDF)
- *JTAG Debugging*
- *Hardware Reference*

For other design documentation for the board, please contact us at sales@espressif.com.

ESP32-DevKitS(-R)

This user guide provides information on ESP32-DevKitS(-R), an ESP32-based flashing board produced by Espressif.

ESP32-DevKitS(-R) is a combination of two board names: ESP32-DevKitS and ESP32-DevKitS-R. S stands for springs, and R stands for WROVER.

ESP32-DevKitS	ESP32-DevKitS-R

The document consists of the following major sections:

- *Getting Started*: Provides an overview of ESP32-DevKitS(-R) and hardware/software setup instructions to get started.
- *Hardware Reference*: Provides more detailed information about ESP32-DevKitS(-R)'s hardware.
- *Related Documents*: Gives links to related documentation.

Getting Started This section describes how to get started with ESP32-DevKitS(-R). It begins with a few introductory sections about ESP32-DevKitS(-R), then Section *How to Flash a Board* provides instructions on how to mount a module onto ESP32-DevKitS(-R), get it ready, and flash firmware onto it.

Overview ESP32-DevKitS(-R) is Espressif's flashing board designed specifically for ESP32. It can be used to flash an ESP32 module without soldering the module to the power supply and signal lines. With a module mounted, ESP32-DevKitS(-R) can also be used as a mini development board like ESP32-DevKitC.

ESP32-DevKitS and ESP32-DevKitS-R boards vary only in layout of spring pins to fit the following ESP32 modules.

- **ESP32-DevKitS:**
  - ESP32-WROOM-32
  - ESP32-WROOM-32D
  - ESP32-WROOM-32U
  - ESP32-SOLO-1
  - ESP32-WROOM-32E
  - ESP32-WROOM-32UE
- **ESP32-DevKitS-R:**
  - ESP32-WROVER (PCB & IPEX)
  - ESP32-WROVER-B (PCB & IPEX)
  - ESP32-WROVER-E
  - ESP32-WROVER-IE

For information about above modules, please refer to ESP32 Series Modules.

Description of Components

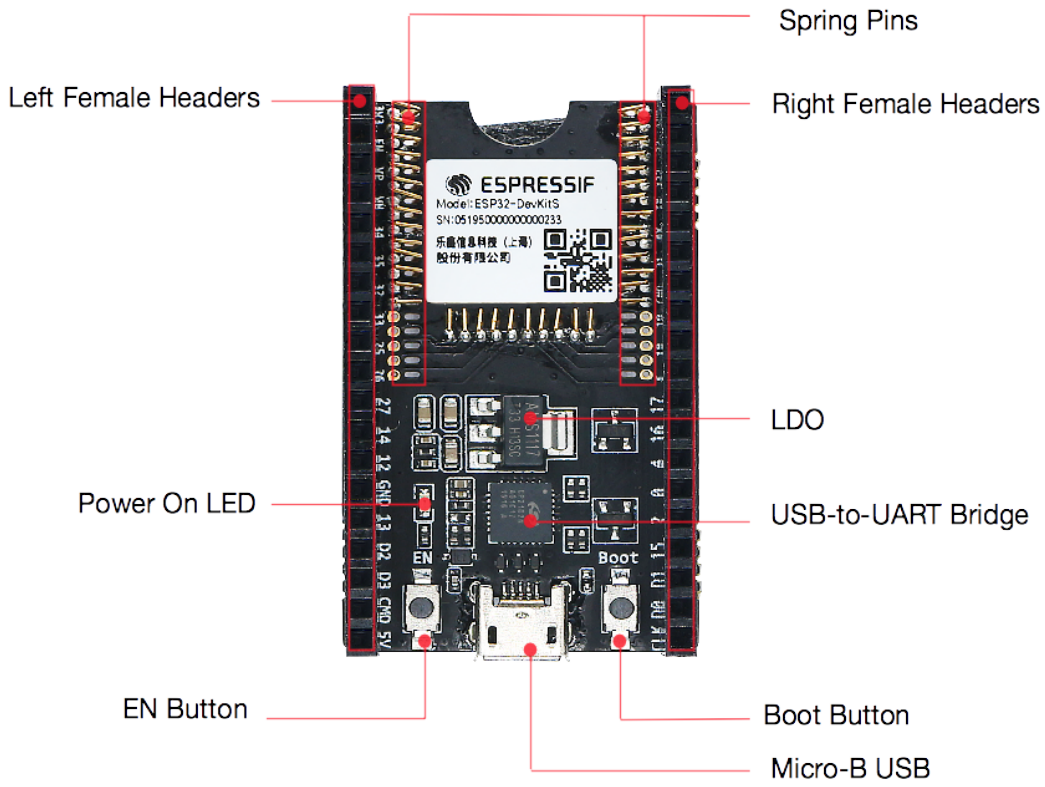


Fig. 37: ESP32-DevKitS - front

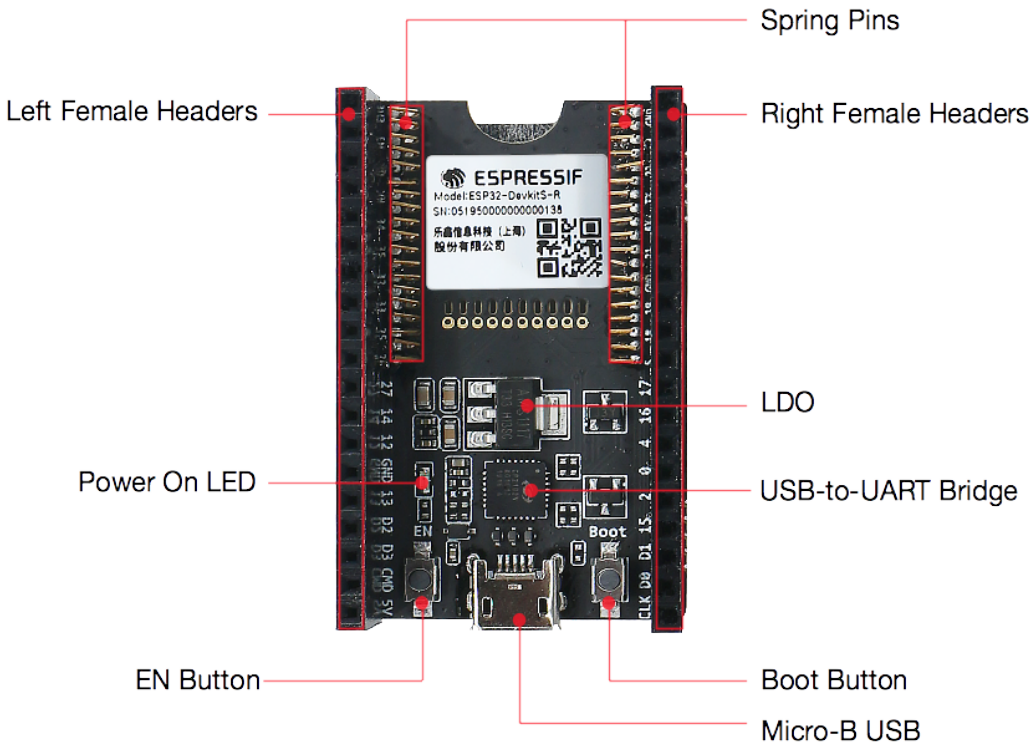


Fig. 38: ESP32-DevKitS-R - front

Key Component	Description
Spring Pins	Click the module in. The pins will fit into the module' s castellated holes.
2.54 mm Female Headers	These female headers are connected to pins of the module mounted on this board. For description of female headers, please refer to *Header Blocks*.
USB-to-UART Bridge	Single-chip USB to UART bridge provides transfer rates of up to 3 Mbps.
LDO	5V-to-3.3V low-dropout voltage regulator (LDO).
Micro-USB Connector/Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
EN Button	Reset button.
Boot Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
Power On LED	Turns on when the USB or power supply is connected to the board.

How to Flash a Board Before powering up your ESP32-DevKitS(-R), please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- An ESP32 module of your choice
- USB 2.0 cable (Standard-A to Micro-B)
- Computer running Windows, Linux, or macOS

Hardware Setup Please mount a module of your choice onto your ESP32-DevKitS(-R) according to the following steps:

- Gently put your module on the ESP32-DevKitS(-R) board. Make sure that castellated holes on your module are aligned with spring pins on the board.
- Press your module down into the board until it clicks.
- Check whether all spring pins are inserted into castellated holes. If there are some misaligned spring pins, place them into castellated holes with tweezers.

Software Setup

Preferred Method The ESP-IDF development framework provides a preferred way of flashing binaries onto ESP32-DevKitS(-R). Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an application example onto your ESP32-DevKitS(-R).

Alternative Method As an alternative, Windows users can flash binaries using the Flash Download Tool. Just download it, unzip it, and follow the instructions inside the *doc* folder.

Note:

1. To flash binary files, ESP32 should be set to Firmware Download mode. This can be done either by the flash tool automatically, or by holding down the Boot button and tapping the EN button.
2. After flashing binary files, the Flash Download Tool restarts your ESP32 module and boots the flashed application by default.

Board Dimensions

Contents and Packaging

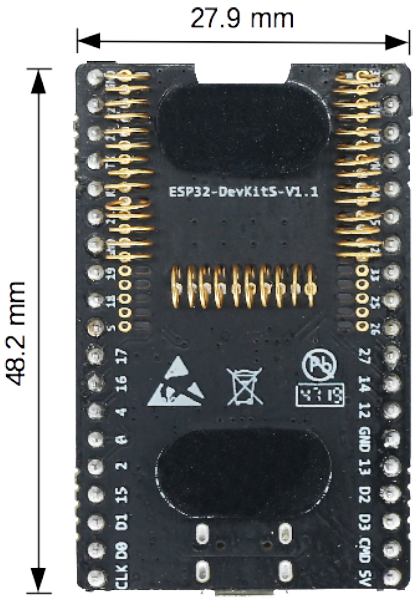


Fig. 39: ESP32-DevKitS board dimensions - back

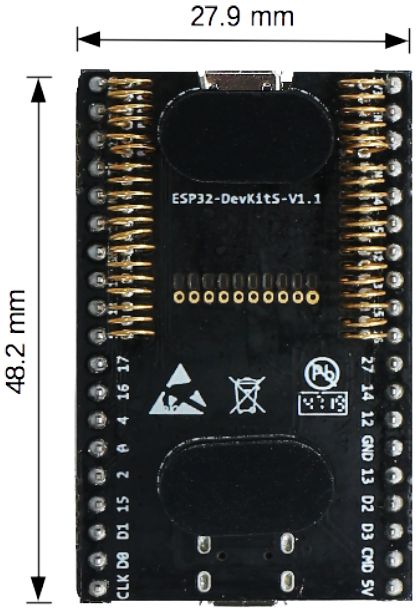


Fig. 40: ESP32-DevKitS-R board dimensions - back

Retail orders If you order a few samples, each ESP32-DevKitS(-R) comes in an individual package in either antistatic bag or any packaging depending on a retailer.

For retail orders, please go to https://www.espressif.com/en/contact-us/get-samples.

Wholesale Orders If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Hardware Reference

Block Diagram A block diagram below shows the components of ESP32-DevKitS(-R) and their interconnections.

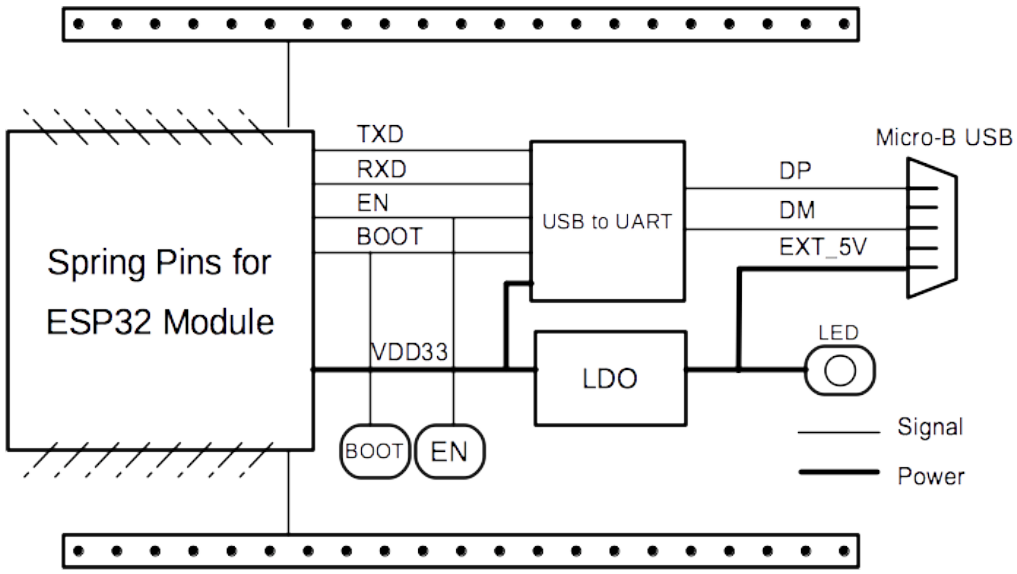


Fig. 41: ESP32-DevKitS(-R) (click to enlarge)

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V and GND header pins
- 3V3 and GND header pins

It is advised to use the first option: micro USB port.

.	Label	Signal
L1	3V3	VDD 3V3
L2	EN	CHIP_PU
L3	VP	SENSOR_VP
L4	VN	SENSOR_VN
L5	34	GPIO34
L6	35	GPIO35
L7	32	GPIO32
L8	33	GPIO33

continues on next page

Table 5 – continued from previous page

.	Label	Signal
L9	25	GPIO25
L10	26	GPIO26
L11	27	GPIO27
L12	14	GPIO14
L13	12	GPIO12
L14	GND	GND
L15	13	GPIO13
L16	D2	SD_DATA2
L17	D3	SD_DATA3
L18	CMD	SD_CMD
L19	5V	External 5V
R1	GND	GND
R2	23	GPIO23
R3	22	GPIO22
R4	TX	U0TXD
R5	RX	U0RXD
R6	21	GPIO21
R7	GND	GND
R8	19	GPIO19
R9	18	GPIO18
R10	5	GPIO5
R11	17	GPIO17
R12	16	GPIO16
R13	4	GPIO4
R14	0	GPIO0
R15	2	GPIO2
R16	15	GPIO15
R17	D1	SD_DATA1
R18	D0	SD_DATA0
R19	CLK	SD_CLK

Header Blocks For the image of header blocks, please refer to *Description of Components*.

Related Documents

- ESP32-DevKitS(-R) Schematics (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROOM-32 Datasheet (PDF)
- ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet (PDF)
- ESP32-SOLO-1 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- ESP32-WROVER-B Datasheet (PDF)
- ESP Product Selector

ESP32-PICO-KIT-1

Overview ESP32-PICO-KIT-1 is an ESP32-based development board produced by Espressif.

The core of this board is ESP32-PICO-V3 - a System-in-Package (SiP) module with complete Wi-Fi and Bluetooth functionalities. Compared to other ESP32 modules, ESP32-PICO-V3 integrates the following peripheral components in one single package, which otherwise would need to be installed separately:

- 40 MHz crystal oscillator
- 4 MB flash
- Filter capacitors
- RF matching network

This setup reduces the costs of additional external components as well as the cost of assembly and testing and also increases the overall usability of the product.

The development board features a USB-to-UART Bridge circuit which allows developers to connect the board to a computer's USB port for flashing and debugging.

All the IO signals and system power on ESP32-PICO-V3 are led out to two rows of 18 x 0.1" header pads on both sides of the development board for easy access. For compatibility with Dupont wires, all header pads are populated with two rows of male pin headers.

Note: ESP32-PICO-KIT-1 comes with male headers by default.

ESP32-PICO-KIT-1 provides the users with hardware for development of applications based on the ESP32, making it easier for users to explore ESP32 functionalities.

This guide covers:

- *Getting Started*: Provides an overview of the ESP32-PICO-KIT-1 and software setup instructions to get started.
- *Contents and Packaging*: Provides information about packaging and contents for retail and wholesale orders.
- *Hardware Reference*: Provides more detailed information about the ESP32-PICO-KIT-1's hardware.
- *Hardware Revision Details*: Covers revision history, known issues, and links to user guides for previous versions of the ESP32-PICO-KIT-1.
- *Related Documents*: Gives links to related documentation.

Getting Started This section describes how to get started with the ESP32-PICO-KIT-1. It begins with a few introductory sections about the ESP32-PICO-KIT-1, then Section *Start Application Development* provides instructions on how to flash firmware onto the ESP32-PICO-KIT-1.

Description of Components The following figure and the table below describe the key components, interfaces, and controls of the ESP32-PICO-KIT-1 board.

Below is the description of the items identified in the figure starting from the top left corner and going clockwise.

Key Component	Description
ESP32-PICO-V3	Standard ESP32-PICO-V3 module soldered to the ESP32-PICO-KIT-1 board. The complete ESP32 system on a chip (ESP32 SoC) has been integrated into the SiP module, requiring only an external antenna with LC matching network, decoupling capacitors, and a pull-up resistor for EN signals to function properly.
LDO	5V-to-3.3V Low dropout voltage regulator (LDO).
USB-to-UART bridge	CP2102N, single-chip USB-to-UART bridge that offers up to 3 Mbps transfers rates.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
5V Power On LED	This red LED turns on when power is supplied to the board. For details, see the schematic in *Related Documents*.
I/O Connector	All the pins on ESP32-PICO-V3 are broken out to pin headers. You can program ESP32 to enable multiple functions, such as PWM, ADC, DAC, I2C, I2S, SPI, etc. For details, please see Section *Pin Descriptions*.
BOOT Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
EN Button	Reset button.

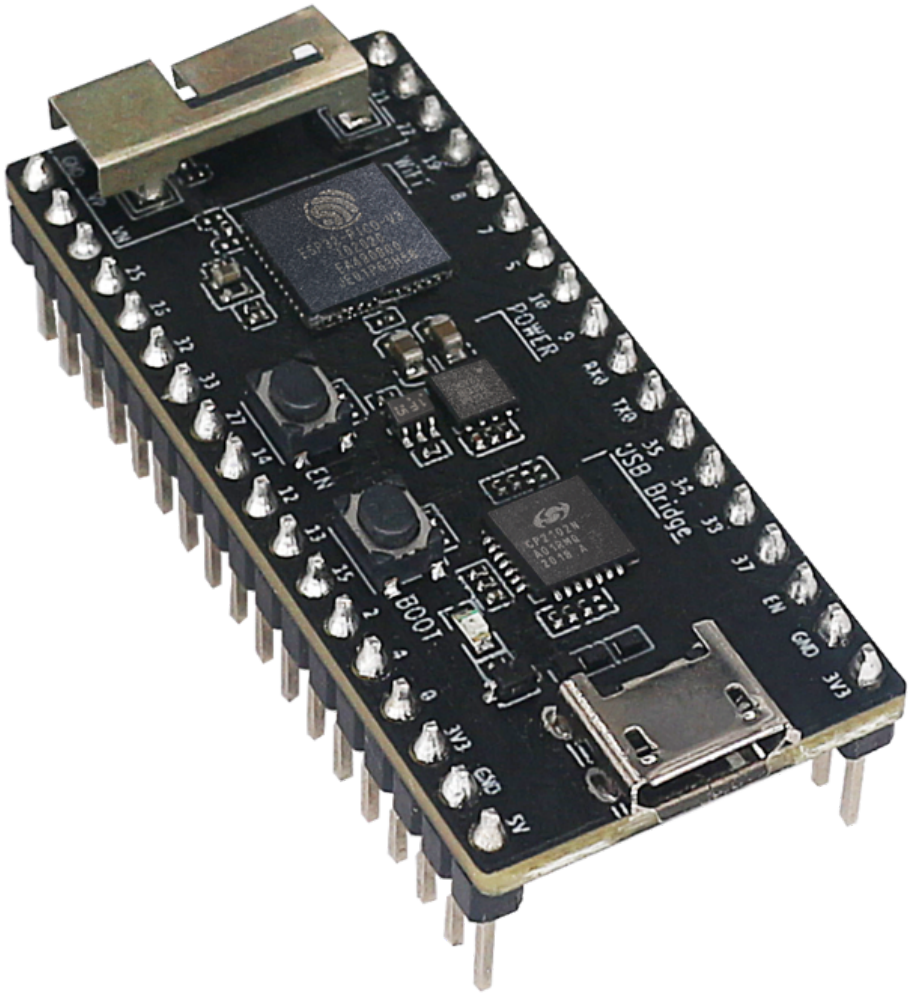


Fig. 42: ESP32-PICO-KIT-1 Overview (click to enlarge)

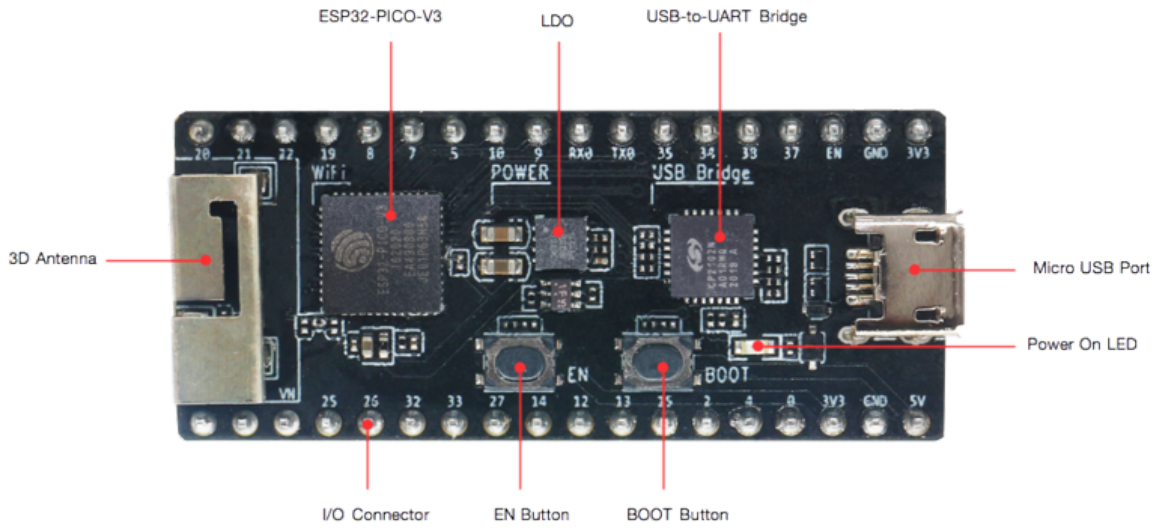


Fig. 43: ESP32-PICO-KIT-1 board layout - front (click to enlarge)

Start Application Development Before powering up your ESP32-PICO-KIT-1, please make sure that the board is in good condition with no obvious signs of damage.

Required Hardware

- 1 x ESP32-PICO-KIT-1
- 1 x USB 2.0 A to Micro B cable
- 1 x Computer running Windows, Linux, or macOS

Software Setup Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment.

Contents and Packaging

Retail Orders If you order one or several samples of the board, each ESP32-PICO-KIT-1 development board comes in an individual package.

For retail orders, please go to https://www.espressif.com/en/contact-us/get-samples.

Wholesale Orders If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Hardware Reference

Block Diagram The block diagram below shows the main components of ESP32-PICO-KIT-1 and their interconnections.

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V/GND header pins
- 3V3/GND header pins

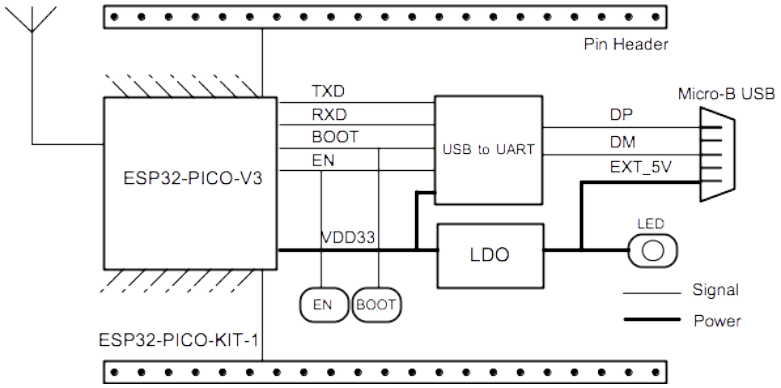


Fig. 44: ESP32-PICO-KIT-1 Block Diagram (click to enlarge)

Warning: The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.

Pin Descriptions The two tables below provide the **Name** and **Function** of I/O header pins on both sides of the board, see *Description of Components*. The pin numbering and header names are the same as in the schematic given in *Related Documents*.

Header J2

No.	Name	Type	Function
1	IO20	I/O	GPIO20
2	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
3	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
4	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
5	IO8	I/O	GPIO8, SD_DATA1, HS1_DATA1, U2CTS
6	IO7	I/O	GPIO7, SD_DATA0, HS1_DATA0, U2RTS
7	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
8	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
9	IO9	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD
10	RXD0	I/O	GPIO3, U0RXD *(See 1)*, CLK_OUT2
11	TXD0	I/O	GPIO1, U0TXD *(See 1)*, CLK_OUT3, EMAC_RXD2
12	IO35	I	ADC1_CH7, RTC_GPIO5
13	IO34	I	ADC1_CH6, RTC_GPIO4
14	IO38	I	GPIO38, ADC1_CH2, RTC_GPIO2
15	IO37	I	GPIO37, ADC1_CH1, RTC_GPIO1
16	EN	I	CHIP_PU
17	GND	P	Ground
18	VDD33 (3V3)	P	3.3 V power supply

Header J3

No.	Name	Type	Function
1	GND	P	Ground
2	SENSOR_VP (FSVP)	I	GPIO36, ADC1_CH0, RTC_GPIO0
3	SENSOR_VN (FSVN)	I	GPIO39, ADC1_CH3, RTC_GPIO3
4	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
5	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
6	IO32	I/O	32K_XP *(See 2a)*, ADC1_CH4, TOUCH9, RTC_GPIO9
7	IO33	I/O	32K_XN *(See 2b)*, ADC1_CH5, TOUCH8, RTC_GPIO8
8	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17, EMAC_RX_DV
9	IO14	I/O	ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2
10	IO12	I/O	ADC2_CH5, TOUCH5, RTC_GPIO15, MTDI *(See 3)*, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3
11	IO13	I/O	ADC2_CH4, TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER
12	IO15	I/O	ADC2_CH3, TOUCH3, RTC_GPIO13, MTDO, HSPICS0, HS2_CMD, SD_CMD, EMAC_RXD3
13	IO2	I/O	ADC2_CH2, TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0
14	IO4	I/O	ADC2_CH0, TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER
15	IO0	I/O	ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK
16	VDD33 (3V3)	P	3.3V power supply
17	GND	P	Ground
18	EXT_5V (5V)	P	5V power supply

Note:

1. This pin is connected to the pin of the USB bridge chip on the board.
2. 32.768 kHz crystal oscillator: a) input b) output
3. The operating voltage of ESP32-PICO-KIT-1' s embedded SPI flash is 3.3 V. Therefore, the strapping pin MTDI should be pulled down during the module power-on reset. If connected, please make sure that this pin is not held up on reset.

Pin Layout

Hardware Revision Details No previous versions available.

Related Documents

- ESP32-PICO-V3 Datasheet (PDF)
- ESP Product Selector
- ESP32-PICO-KIT-1 Schematic (PDF)
- ESP32-PICO-KIT-1 PCB Layout (PDF)

For other design documentation for the board, please contact us at sales@espressif.com.

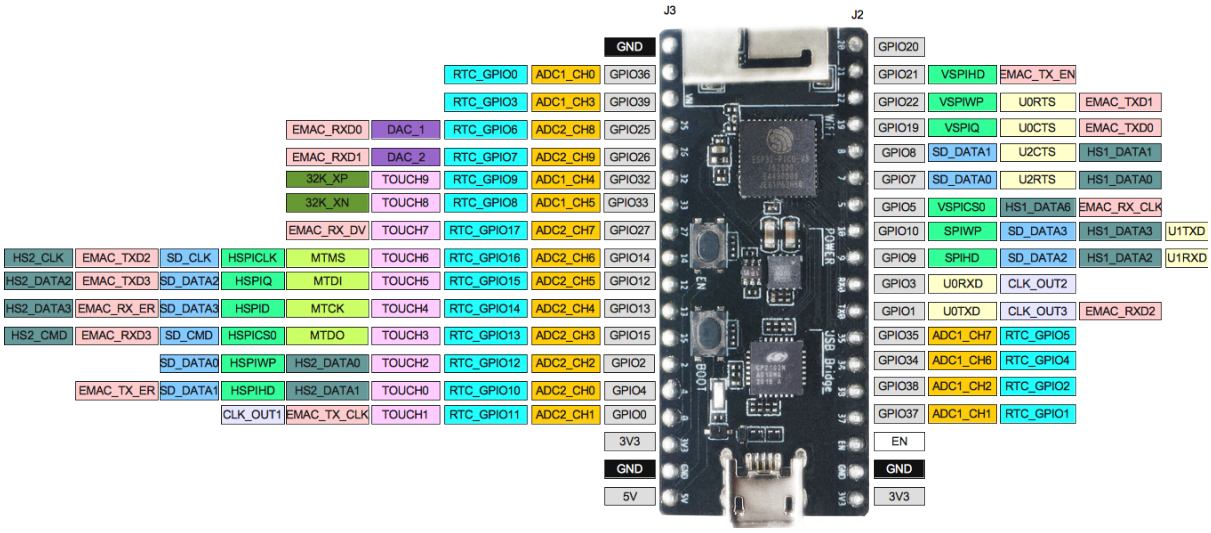


Fig. 45: ESP32-PICO-KIT-1 Pin Layout(click to enlarge)

ESP32-PICO-DevKitM-2

Overview ESP32-PICO-DevKitM-2 is an ESP32-based development board produced by Espressif.

The core of this board is ESP32-PICO-MINI-02(02U) module with complete Wi-Fi and Bluetooth functionalities. The development board features a USB-to-UART Bridge circuit which allows developers to connect the board to a computer's USB port for flashing and debugging.

All the IO signals and system power on ESP32-PICO-MINI-02(02U) are led out to two rows of 18 x 0.1" header pads on both sides of the development board for easy access. For compatibility with Dupont wires, all header pads are populated with two rows of male pin headers.

Note: ESP32-PICO-DevKitM-2 comes with male headers by default.

ESP32-PICO-DevKitM-2 provides the users with hardware for development of applications based on the ESP32, making it easier for users to explore ESP32 functionalities.

This guide covers:

- *Getting Started*: Provides an overview of the ESP32-PICO-DevKitM-2 and software setup instructions to get started.
- *Contents and Packaging*: Provides information about packaging and contents for retail and wholesale orders.
- *Hardware Reference*: Provides more detailed information about the ESP32-PICO-DevKitM-2's hardware.
- *Hardware Revision Details*: Covers revision history, known issues, and links to user guides for previous versions (if any) of the ESP32-PICO-DevKitM-2.
- *Related Documents*: Gives links to related documentation.

Getting Started This section describes how to get started with the ESP32-PICO-DevKitM-2. It begins with a few introductory sections about the ESP32-PICO-DevKitM-2, then Section *Start Application Development* provides instructions on how to flash firmware onto the ESP32-PICO-DevKitM-2.

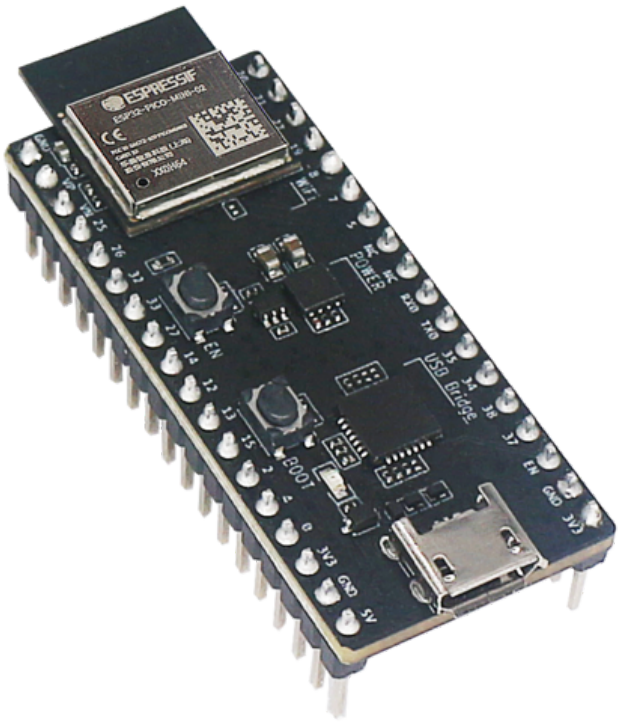


Fig. 46: ESP32-PICO-DevKitM-2 Overview (click to enlarge)

Description of Components The following figure and the table below describe the key components, interfaces, and controls of the ESP32-PICO-DevKitM-2 board. We take the board with a ESP32-PICO-MINI-02 module as an example in the following sections.

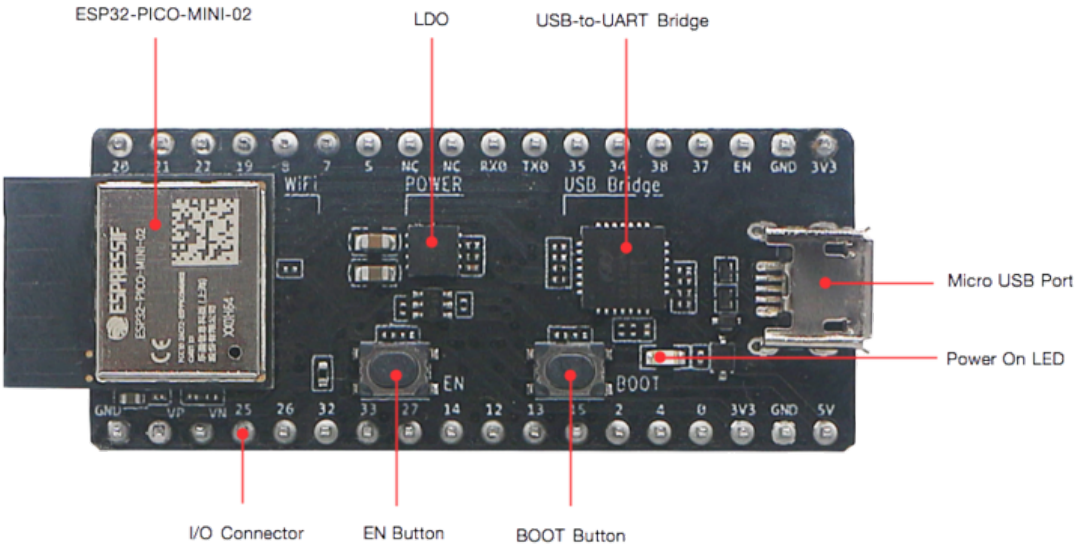


Fig. 47: ESP32-PICO-DevKitM-2 board layout - front (click to enlarge)

Below is the description of the items identified in the figure starting from the top left corner and going clockwise.

Key Component	Description
ESP32-PICO-MINI-02	Standard ESP32-PICO-MINI-02 module soldered to the ESP32-PICO-DevKitM-2 board. The complete ESP32 system on a chip (ESP32 SoC) has been integrated into the module. Users can also select the board with ESP32-PICO-MINI-02U soldered.
LDO	V-to-3.3V Low dropout voltage regulator (LDO).
USB-to-UART bridge	CP2102N, single-chip USB-UART bridge that offers up to 3 Mbps transfers rates.
Micro-B USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the board.
5V Power On LED	This red LED turns on when power is supplied to the board. For details, see the schematic in *Related Documents*.
I/O Connector	All the pins on ESP32-PICO-MINI-02 are broken out to pin headers. You can program ESP32 to enable multiple functions, such as PWM, ADC, DAC, I2C, I2S, SPI, etc. For details, please see Section *Pin Descriptions*.
BOOT Button	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
EN Button	Reset button.

Start Application Development Before powering up your ESP32-PICO-DevKitM-2, please make sure that the board is in good condition with no obvious signs of damage.

Required Hardware

- 1 x ESP32-PICO-DevKitM-2
- 1 x USB 2.0 A to Micro B cable
- 1 x Computer running Windows, Linux, or macOS

Software Setup Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment.

Contents and Packaging

Retail Orders If you order one or several samples of the board, each ESP32-PICO-DevKitM-2 development board comes in an individual package.

For retail orders, please go to https://www.espressif.com/en/contact-us/get-samples.

Wholesale Orders If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Hardware Reference

Block Diagram The block diagram below shows the main components of ESP32-PICO-DevKitM-2 and their interconnections.

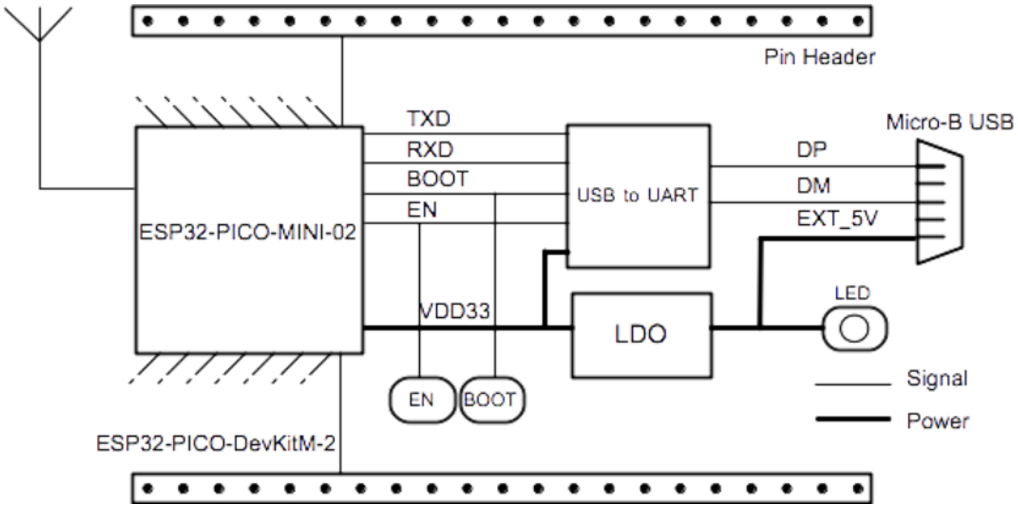


Fig. 48: ESP32-PICO-DevKitM-2 Block Diagram (click to enlarge)

Power Supply Options There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V/GND header pins
- 3V3/GND header pins

> **Warning:** The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.

Pin Descriptions The two tables below provide the **Name** and **Function** of I/O header pins on both sides of the board, see *Description of Components*. The pin numbering and header names are the same as in the schematic given in *Related Documents*.

Header J2

No.	Name	Type	Function
1	IO20	I/O	GPIO20
2	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
3	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
4	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
5	IO8	I/O	GPIO8, SD_DATA1, HS1_DATA1, U2CTS
6	IO7	I/O	GPIO7, SD_DATA0, HS1_DATA0, U2RTS
7	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
8	NC	-	NC
9	NC	-	NC
10	RXD0	I/O	GPIO3, U0RXD *(See 1)*, CLK_OUT2
11	TXD0	I/O	GPIO1, U0TXD *(See 1)*, CLK_OUT3, EMAC_RXD2
12	IO35	I	ADC1_CH7, RTC_GPIO5
13	IO34	I	ADC1_CH6, RTC_GPIO4
14	IO38	I	GPIO38, ADC1_CH2, RTC_GPIO2
15	IO37	I	GPIO37, ADC1_CH1, RTC_GPIO1
16	EN	I	CHIP_PU
17	GND	P	Ground
18	VDD33 (3V3)	P	3.3 V power supply

Header J3

No.	Name	Type	Function
1	GND	P	Ground
2	SENSOR_VP (FSVP)	I	GPIO36, ADC1_CH0, RTC_GPIO0
3	SENSOR_VN (FSVN)	I	GPIO39, ADC1_CH3, RTC_GPIO3
4	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
5	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
6	IO32	I/O	32K_XP *(See 2a)*, ADC1_CH4, TOUCH9, RTC_GPIO9
7	IO33	I/O	32K_XN *(See 2b)*, ADC1_CH5, TOUCH8, RTC_GPIO8
8	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17, EMAC_RX_DV
9	IO14	I/O	ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2
10	IO12	I/O	ADC2_CH5, TOUCH5, RTC_GPIO15, MTDI *(See 3)*, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3
11	IO13	I/O	ADC2_CH4, TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER
12	IO15	I/O	ADC2_CH3, TOUCH3, RTC_GPIO13, MTDO, HSPICS0, HS2_CMD, SD_CMD, EMAC_RXD3
13	IO2	I/O	ADC2_CH2, TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0
14	IO4	I/O	ADC2_CH0, TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER
15	IO0	I/O	ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK
16	VDD33 (3V3)	P	3.3V power supply
17	GND	P	Ground
18	EXT_5V (5V)	P	5V power supply

Note:

1. This pin is connected to the pin of the USB bridge chip on the board.
2. 32.768 kHz crystal oscillator: a) input b) output
3. The operating voltage of ESP32-PICO-DevKitM-2' s embedded SPI flash is 3.3 V. Therefore, the strapping pin MTDI should be pulled down during the module power-on reset. If connected, please make sure that this pin is not held up on reset.

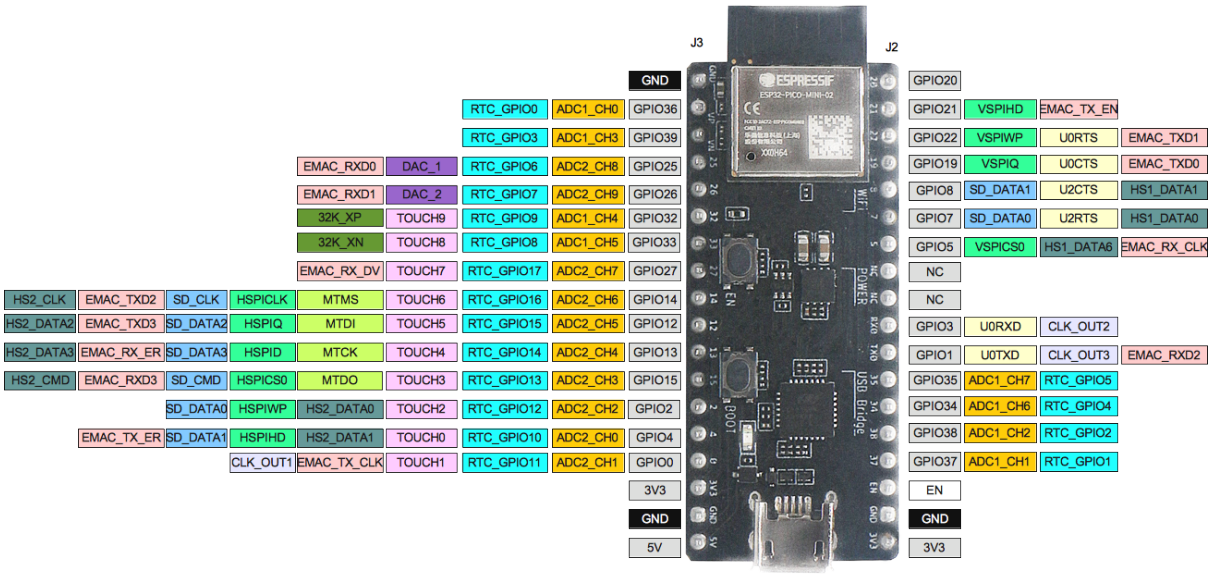


Fig. 49: ESP32-PICO-DevKitM-2 Pin Layout (click to enlarge)

Pin Layout

Hardware Revision Details

No previous versions available.

Related Documents

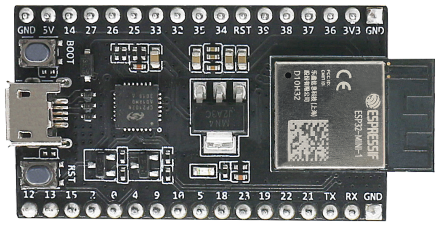
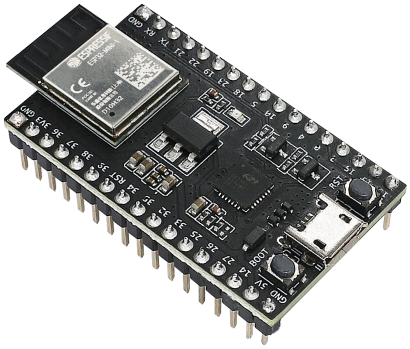
- ESP32-PICO-MINI-02 & ESP32-PICO-MINI-1U Datasheet (PDF)
- ESP Product Selector
- ESP32-PICO-DevKitM-2 Schematic (PDF)
- ESP32-PICO-DevKitM-2 PCB Layout (PDF)

For other design documentation for the board, please contact us at sales@espressif.com.

ESP32-DevKitM-1

This user guide will help you get started with ESP32-DevKitM-1 and will also provide more in-depth information.

ESP32-DevKitM-1 is an ESP32-MINI-1(1U)-based development board produced by Espressif. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Users can either connect peripherals with jumper wires or mount ESP32-DevKitM-1 on a breadboard.

	
ESP32-DevKitM-1 - front	ESP32-DevKitM-1 - isometric

The document consists of the following major sections:

- *Getting started*: Provides an overview of the ESP32-DevKitM-1 and hardware/software setup instructions to get started.
- *Hardware reference*: Provides more detailed information about the ESP32-DevKitM-1 ' s hardware.
- *Related Documents*: Gives links to related documentaiton.

Getting Started This section describes how to get started with ESP32-DevKitM-1. It begins with a few introductory sections about the ESP32-DevKitM-1, then Section *Start Application Development* provides instructions on how to do the initial hardware setup and then how to flash firmware onto the ESP32-DevKitM-1.

Overview This is a small and convenient development board that features:

- ESP32-MINI-1, or ESP32-MINI-1U module
- USB-to-serial programming interface that also provides power supply for the board
- pin headers
- pushbuttons for reset and activation of Firmware Download mode
- a few other components

Contents and Packaging

Retail orders If you order a few samples, each ESP32-DevKitM-1 comes in an individual package in either anti-static bag or any packaging depending on your retailer.

For retail orders, please go to https://www.espressif.com/en/contact-us/get-samples.

Wholesale Orders If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Description of Components The following figure and the table below describe the key components, interfaces and controls of the ESP32-DevKitM-1 board. We take the board with a ESP32-MINI-1 module as an example in the following sections.

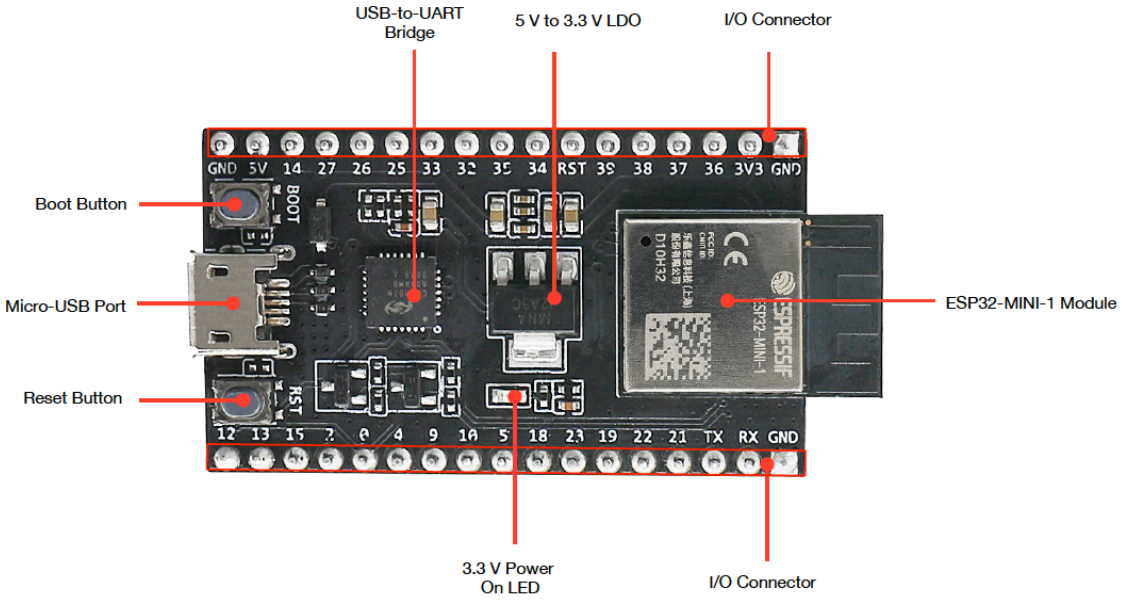


Fig. 50: ESP32-DevKitM-1 - front

Key Component	Description
On-board module	ESP32-MINI-1 module or ESP32-MINI-1U module. ESP32-MINI-1 comes with an on-board PCB antenna. ESP32-MINI-1U comes with an external antenna connector. The two modules both have a 4 MB flash in chip package. For details, please see ESP32-MINI-1 & ESP32-MINI-1U Datasheet.
5 V to 3.3 V LDO	Power regulator converts 5 V to 3.3 V.
Boot Button	Download button. Holding down **Boot** and then pressing **Reset** initiates Firmware Download mode for downloading firmware through the serial port.
Reset Button	Reset Button
Micro-USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32 chip.
USB-to-UART Bridge	Single USB-UART bridge chip provides transfer rates up to 3 Mbps.
3.3 V Power On LED	Turns on when the USB is connected to the board. For details, please see the schematics in *Related Documents*.
I/O Connector	All available GPIO pins (except for the SPI bus for flash) are broken out to the pin headers on the board. Users can program ESP32 chip to enable multiple functions.

Start Application Development Before powering up your ESP32-DevKitM-1, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- ESP32-DevKitM-1
- USB 2.0 cable (Standard-A to Micro-B)
- Computer running Windows, Linux, or macOS

Software Setup Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an application example onto your ESP32-DevKitM-1.

Attention: ESP32-DevKitM-1 boards manufactured before December 2, 2021 have a single core module installed. To verify what module you have, please check module marking information in PCN-2021-021 . If your board has a single core module installed, please enable single core mode (*CONFIG_FREERTOS_UNICORE*) in *menuconfig* before flashing your applications.

Hardware Reference

Block Diagram A block diagram below shows the components of ESP32-DevKitM-1 and their interconnections.

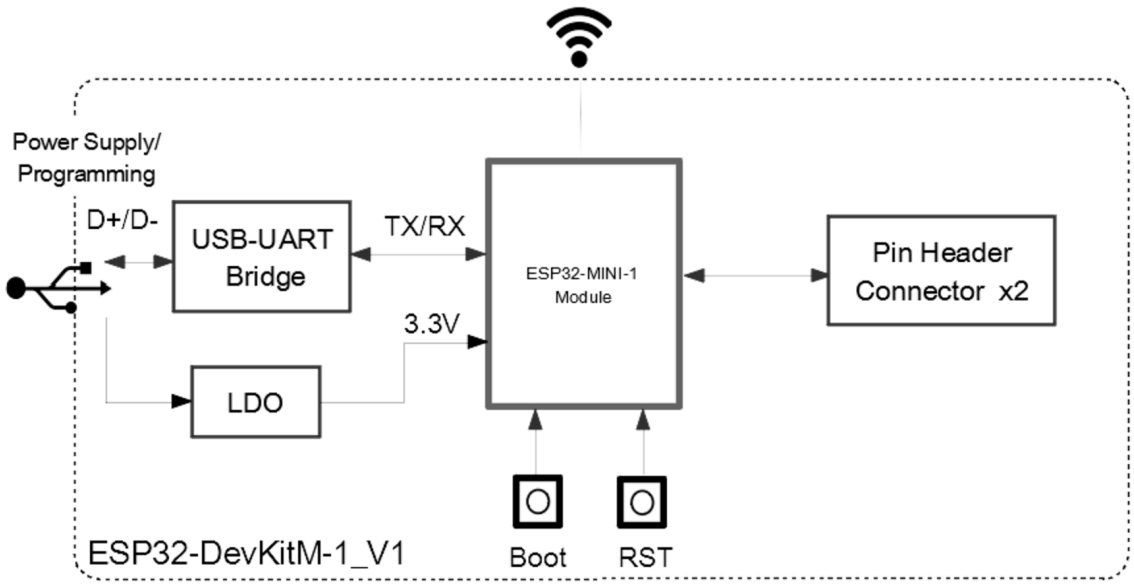


Fig. 51: ESP32-DevKitM-1

Power Source Select There are three mutually exclusive ways to provide power to the board:

- Micro USB port, default power supply
- 5V and GND header pins
- 3V3 and GND header pins

Warning:

- The power supply must be provided using **one and only one of the options above**, otherwise the board and/or the power supply source can be damaged.
- Power supply by micro USB port is recommended.

Pin Descriptions The table below provides the Name and Function of pins on both sides of the board. For peripheral pin configurations, please refer to ESP32 Datasheet.

No.	Name	Type	Function
1	GND	P	Ground
2	3V3	P	3.3 V power supply

continues on next page

Table 6 – continued from previous page

No.	Name	Type	Function
3	I36	I	GPIO36, ADC1_CH0, RTC_GPIO0
4	I37	I	GPIO37, ADC1_CH1, RTC_GPIO1
5	I38	I	GPIO38, ADC1_CH2, RTC_GPIO2
6	I39	I	GPIO39, ADC1_CH3, RTC_GPIO3
7	RST	I	Reset; High: enable; Low: powers off
8	I34	I	GPIO34, ADC1_CH6, RTC_GPIO4
9	I35	I	GPIO35, ADC1_CH7, RTC_GPIO5
10	IO32	I/O	GPIO32, XTAL_32K_P (32.768 kHz crystal oscillator input), ADC1_CH4, TOUCH9, RTC_GPIO9
11	IO33	I/O	GPIO33, XTAL_32K_N (32.768 kHz crystal oscillator output), ADC1_CH5, TOUCH8, RTC_GPIO8
12	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
13	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
14	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17, EMAC_RX_DV
15	IO14	I/O	GPIO14, ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2
16	5V	P	5 V power supply
17	IO12	I/O	GPIO12, ADC2_CH5, TOUCH5, RTC_GPIO15, MTDI, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3
18	IO13	I/O	GPIO13, ADC2_CH4, TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER
19	IO15	I/O	GPIO15, ADC2_CH3, TOUCH3, RTC_GPIO13, MTDO, HSPICS0, HS2_CMD, SD_CMD, EMAC_RXD3
20	IO2	I/O	GPIO2, ADC2_CH2, TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0
21	IO0	I/O	GPIO0, ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK
22	IO4	I/O	GPIO4, ADC2_CH0, TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER
23	IO9	I/O	GPIO9, HS1_DATA2, U1RXD, SD_DATA2
24	IO10	I/O	GPIO10, HS1_DATA3, U1TXD, SD_DATA3
25	IO5	I/O	GPIO5, HS1_DATA6, VSPICS0, EMAC_RX_CLK
26	IO18	I/O	GPIO18, HS1_DATA7, VSPICLK
27	IO23	I/O	GPIO23, HS1_STROBE, VSPID
28	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
29	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
30	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
31	TXD0	I/O	GPIO1, U0TXD, CLK_OUT3, EMAC_RXD2
32	RXD0	I/O	GPIO3, U0RXD, CLK_OUT2

Hardware Revision Details No previous versions available.

Related Documents

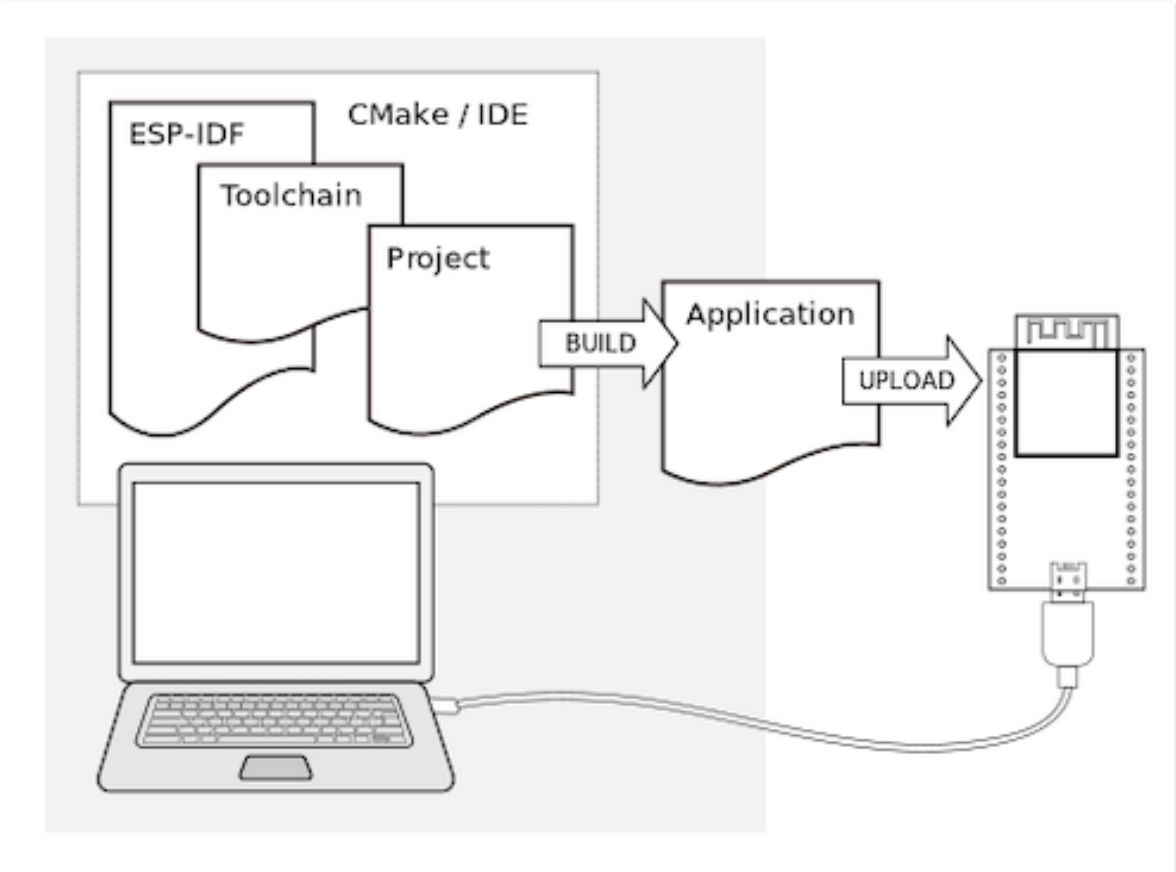
- ESP32-MINI-1 & ESP32-MINI-1U Datasheet (PDF)
- ESP32-DevKitM-1 Schematics (PDF)
- ESP32-DevKitM-1 PCB layout (PDF)
- ESP32-DevKitM-1 layout (DXF) - You can view it with Autodesk Viewer online
- ESP32 Datasheet (PDF)
- ESP Product Selector

For other design documentation for the board, please contact us at sales@espressif.com.

1.2.2 Software

To start using ESP-IDF on **ESP32**, install the following software:

- **Toolchain** to compile code for ESP32
- **Build tools** - CMake and Ninja to build a full **Application** for ESP32
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the **Toolchain**



1.3 Installation

To install all the required software, we offer some different ways to facilitate this task. Choose from one of the available options.

1.3.1 IDE

Note: We highly recommend installing the ESP-IDF through your favorite IDE.

- Eclipse Plugin
- VSCode Extension

1.3.2 Manual Installation

For the manual procedure, please select according to your operating system.

Standard Setup of Toolchain for Windows

Introduction ESP-IDF requires some prerequisite tools to be installed so you can build firmware for supported chips. The prerequisite tools include Python, Git, cross-compilers, CMake and Ninja build tools.

For this Getting Started we're going to use the Command Prompt, but after ESP-IDF is installed you can use Eclipse Plugin or another graphical IDE with CMake support instead.

Note: Limitations: - The installation path of ESP-IDF and ESP-IDF Tools must not be longer than 90 characters. Too long installation paths might result in a failed build. - The installation path of Python or ESP-IDF must not contain white spaces or parentheses. - The installation path of Python or ESP-IDF should not contain special characters (non-ASCII) unless the operating system is configured with "Unicode UTF-8" support.

System Administrator can enable the support via Control Panel - Change date, time, or number formats - Administrative tab - Change system locale - check the option "Beta: Use Unicode UTF-8 for worldwide language support" - Ok and reboot the computer.

ESP-IDF Tools Installer The easiest way to install ESP-IDF's prerequisites is to download one of ESP-IDF Tools Installers.

Windows Installer Download

What is the usecase for Online and Offline Installer Online Installer is very small and allows the installation of all available releases of ESP-IDF. The installer will download only necessary dependencies including Git For Windows during the installation process. The installer stores downloaded files in the cache directory `%userprofile%\.espressif`

Offline Installer does not require any network connection. The installer contains all required dependencies including Git For Windows .

Components of the installation The installer deploys the following components:

- Embedded Python
- Cross-compilers
- OpenOCD
- CMake and Ninja build tools
- ESP-IDF

The installer also allows reusing the existing directory with ESP-IDF. The recommended directory is `%userprofile%\Desktop\esp-idf` where `%userprofile%` is your home directory.

Launching ESP-IDF Environment At the end of the installation process you can check out option `Run ESP-IDF PowerShell Environment` or `Run ESP-IDF Command Prompt (cmd.exe)`. The installer will launch ESP-IDF environment in selected prompt.

`Run ESP-IDF PowerShell Environment`:

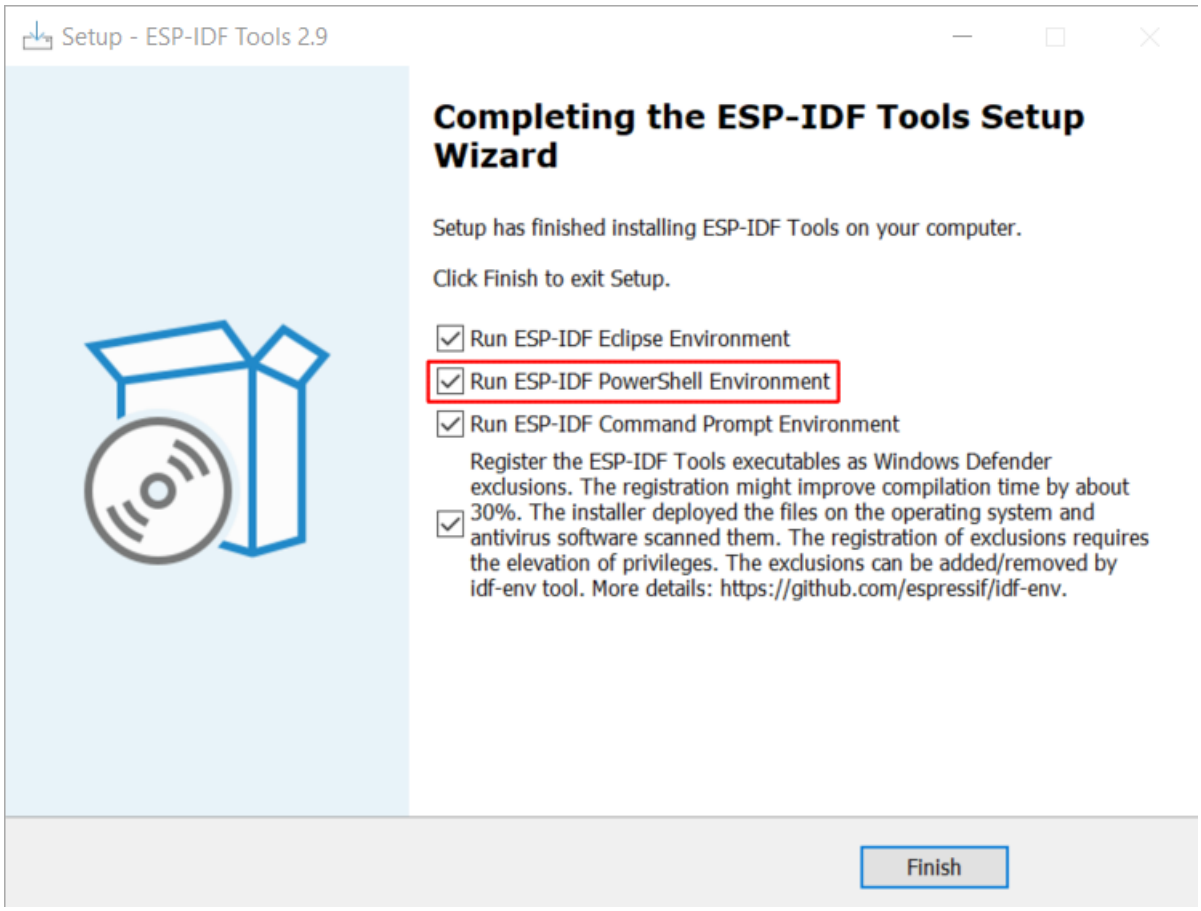


Fig. 52: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF PowerShell Environment

`Run ESP-IDF Command Prompt (cmd.exe)`:

Using the Command Prompt For the remaining Getting Started steps, we ' re going to use the Windows Command Prompt.

ESP-IDF Tools Installer also creates a shortcut in the Start menu to launch the ESP-IDF Command Prompt. This shortcut launches the Command Prompt (cmd.exe) and runs `export.bat` script to set up the environment variables (`PATH`, `IDF_PATH` and others). Inside this command prompt, all the installed tools are available.

Note that this shortcut is specific to the ESP-IDF directory selected in the ESP-IDF Tools Installer. If you have multiple ESP-IDF directories on the computer (for example, to work with different versions of ESP-IDF), you have two options to use them:

1. Create a copy of the shortcut created by the ESP-IDF Tools Installer, and change the working directory of the new shortcut to the ESP-IDF directory you wish to use.
2. Alternatively, run `cmd.exe`, then change to the ESP-IDF directory you wish to use, and run `export.bat`. Note that unlike the previous option, this way requires Python and Git to be present in `PATH`. If you get errors related to Python or Git not being found, use the first option.

First Steps on ESP-IDF Now since all requirements are met, the next topic will guide you on how to start your first project.

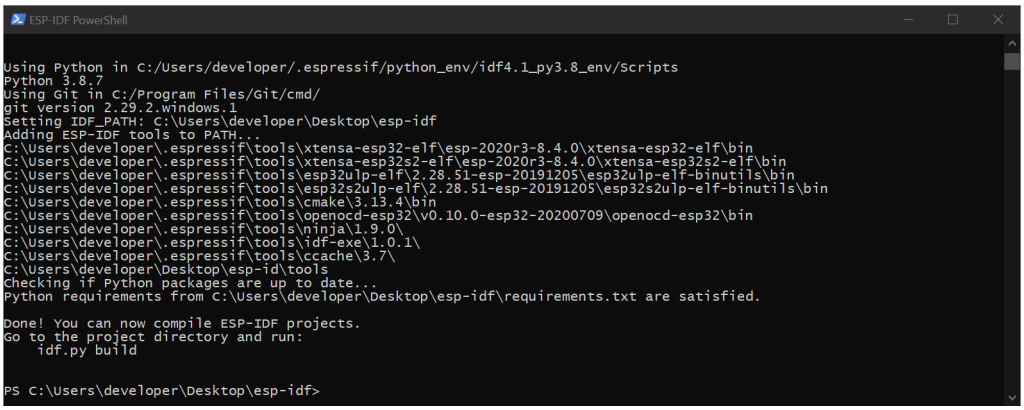


Fig. 53: ESP-IDF PowerShell

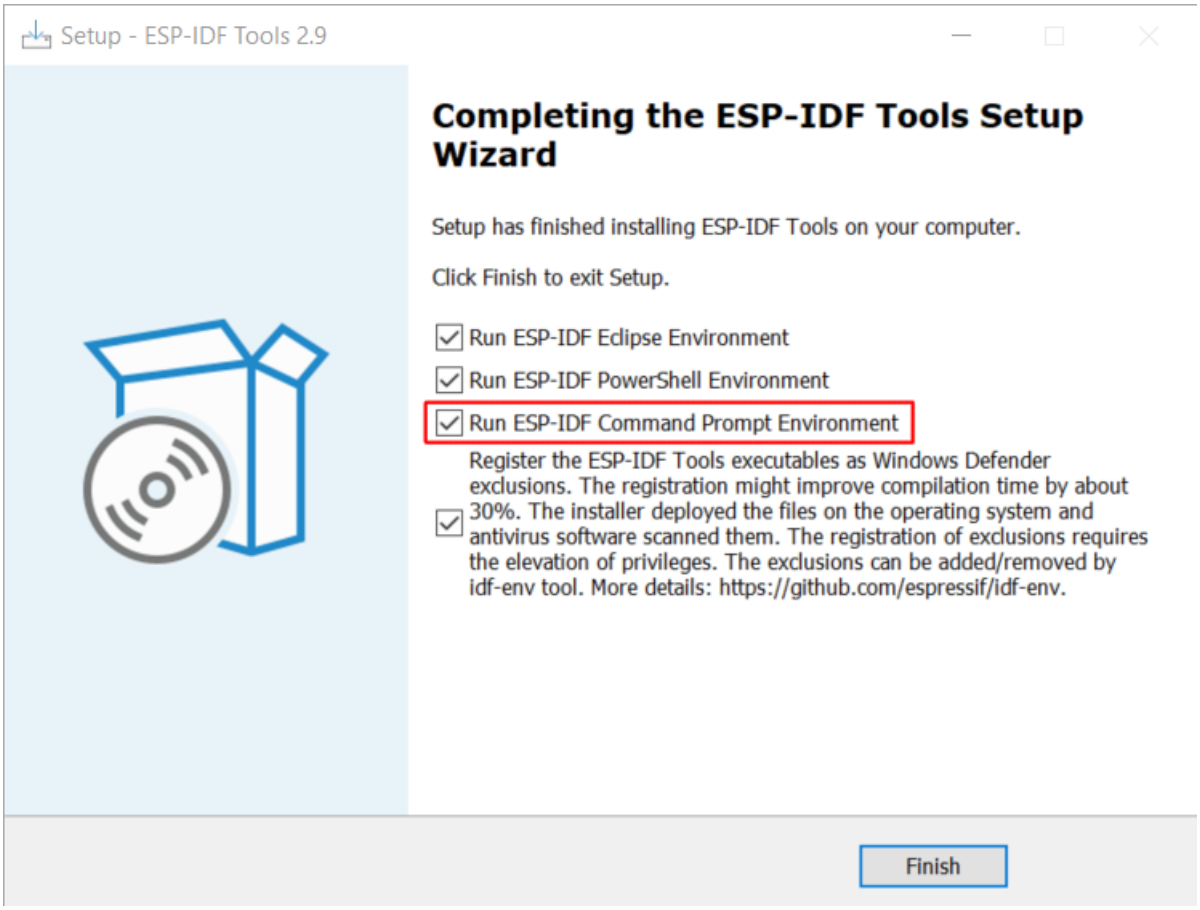


Fig. 54: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF Command Prompt (cmd.exe)

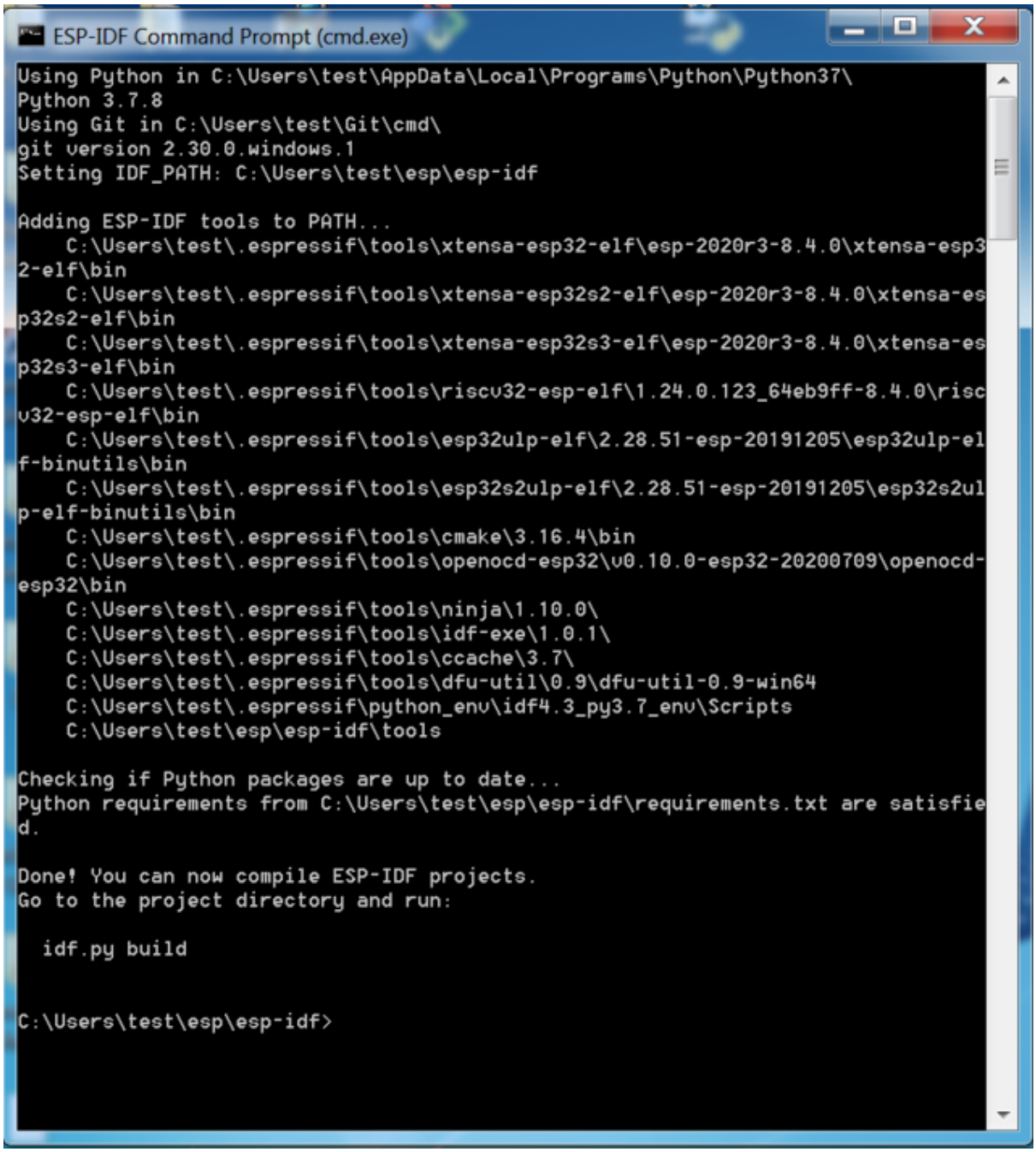


Fig. 55: ESP-IDF Command Prompt

This guide will help you on the first steps using ESP-IDF. Follow this guide to start a new project on the ESP32 and build, flash, and monitor the device output.

Note: If you have not yet installed ESP-IDF, please go to *Installation* and follow the instruction in order to get all the software needed to use this guide.

Start a Project Now you are ready to prepare your application for ESP32. You can start with get-started/hello_world project from examples directory in ESP-IDF.

Important: The ESP-IDF build system does not support spaces in the paths to either ESP-IDF or to projects.

Copy the project get-started/hello_world to `~/esp` directory:

```
cd %userprofile%\esp
xcopy /e /i %IDF_PATH%\examples\get-started\hello_world hello_world
```

Note: There is a range of example projects in the examples directory in ESP-IDF. You can copy any project in the same way as presented above and run it. It is also possible to build examples in-place without copying them first.

Connect Your Device Now connect your ESP32 board to the computer and check under which serial port the board is visible.

Serial port names start with `COM` in Windows.

If you are not sure how to check the serial port name, please refer to *Establish Serial Connection with ESP32* for full details.

Note: Keep the port name handy as you will need it in the next steps.

Configure Your Project Navigate to your `hello_world` directory, set ESP32 as the target, and run the project configuration utility `menuconfig`.

Windows

```
cd %userprofile%\esp\hello_world
idf.py set-target esp32
idf.py menuconfig
```

After opening a new project, you should first set the target with `idf.py set-target esp32`. Note that existing builds and configurations in the project, if any, will be cleared and initialized in this process. The target may be saved in the environment variable to skip this step at all. See *Select the Target Chip: set-target* for additional information.

If the previous steps have been done correctly, the following menu appears:

You are using this menu to set up project specific variables, e.g., Wi-Fi network name and password, the processor speed, etc. Setting up the project with menuconfig may be skipped for "hello_word", since this example runs with default configuration.

Attention: If you use ESP32-DevKitC board with the **ESP32-SOLO-1** module, or ESP32-DevKitM-1 board with the **ESP32-MIN1-1(1U)** module, please enable single core mode (*CONFIG_FREERTOS_UNICORE*) in menuconfig before flashing examples.

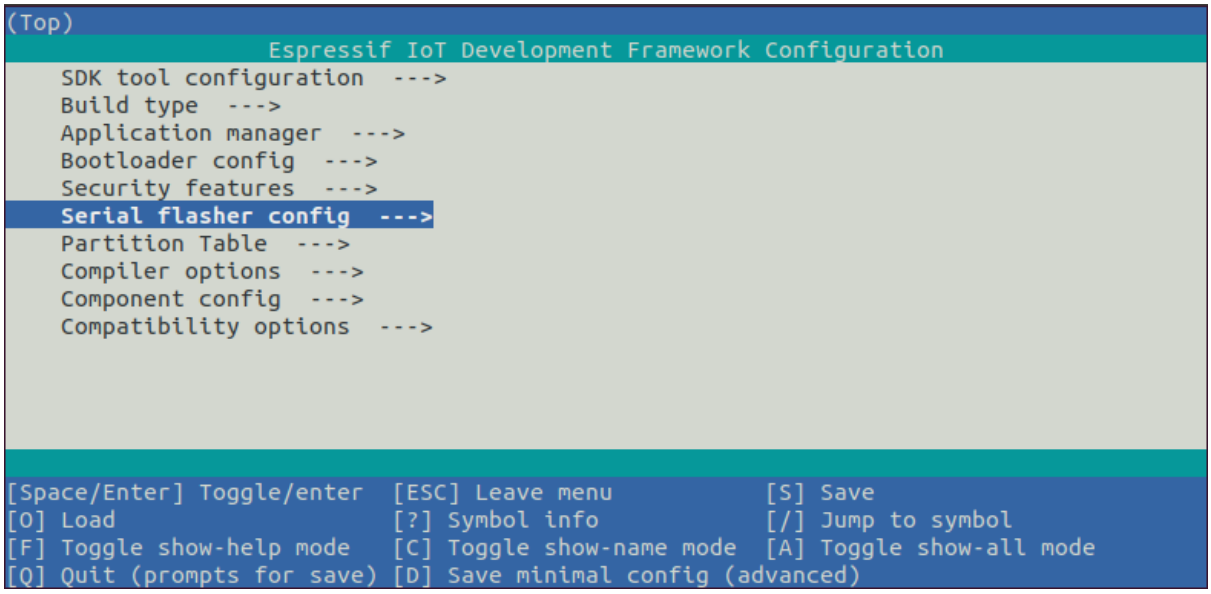


Fig. 56: Project configuration - Home window

Note: The colors of the menu could be different in your terminal. You can change the appearance with the option `--style`. Please run `idf.py menuconfig --help` for further information.

If you are using one of the supported development boards, you can speed up your development by using Board Support Package. See *Additional Tips* for more information.

Build the Project Build the project by running:

```
idf.py build
```

This command will compile the application and all ESP-IDF components, then it will generate the bootloader, partition table, and application binaries.

```
$ idf.py build
Running cmake in directory /path/to/hello_world/build
Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
Warn about uninitialized values.
-- Found Git: /usr/bin/git (found version "2.17.0")
-- Building empty aws_iot component due to configuration
-- Component names: ...
-- Component paths: ...

... (more lines of build system output)

[527/527] Generating hello_world.bin
esptool.py v2.3.1

Project build complete. To flash, run this command:
../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600 write_flash -
↪-flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/hello_world.
↪bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/partition_table/
↪partition-table.bin
or run 'idf.py -p PORT flash'
```

If there are no errors, the build will finish by generating the firmware binary .bin files.

Flash onto the Device Flash the binaries that you just built (bootloader.bin, partition-table.bin and hello_world.bin) onto your ESP32 board by running:

```
idf.py -p PORT [-b BAUD] flash
```

Replace PORT with your ESP32 board's serial port name.

You can also change the flasher baud rate by replacing BAUD with the baud rate you need. The default baud rate is `460800`.

For more information on idf.py arguments, see *idf.py*.

Note: The option `flash` automatically builds and flashes the project, so running `idf.py build` is not necessary.

Encountered Issues While Flashing? If you run the given command and see errors such as "Failed to connect", there might be several reasons for this. One of the reasons might be issues encountered by `esptool.py`, the utility that is called by the build system to reset the chip, interact with the ROM bootloader, and flash firmware. One simple solution to try is manual reset described below, and if it does not help you can find more details about possible issues in Troubleshooting.

`esptool.py` resets ESP32 automatically by asserting DTR and RTS control lines of the USB to serial converter chip, i.e., FTDI or CP210x (for more information, see *Establish Serial Connection with ESP32*). The DTR and RTS control lines are in turn connected to `GPIO0` and `CHIP_PU` (EN) pins of ESP32, thus changes in the voltage levels of DTR and RTS will boot ESP32 into Firmware Download mode. As an example, check the schematic for the ESP32 DevKitC development board.

In general, you should have no problems with the official esp-idf development boards. However, `esptool.py` is not able to reset your hardware automatically in the following cases:

- Your hardware does not have the DTR and RTS lines connected to `GPIO0` and `CHIP_PU`
- The DTR and RTS lines are configured differently
- There are no such serial control lines at all

Depending on the kind of hardware you have, it may also be possible to manually put your ESP32 board into Firmware Download mode (reset).

- For development boards produced by Espressif, this information can be found in the respective getting started guides or user guides. For example, to manually reset an ESP-IDF development board, hold down the **Boot** button (`GPIO0`) and press the **EN** button (`CHIP_PU`).
- For other types of hardware, try pulling `GPIO0` down.

Normal Operation When flashing, you will see the output log similar to the following:

```
...
esptool.py --chip esp32 -p /dev/ttyUSB0 -b 460800 --before=default_reset --
↪after=hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB␣
↪0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.bin␣
↪0x10000 hello_world.bin
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting........_
Chip is ESP32D0WDQ6 (revision 0)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 24:0a:c4:05:b9:14
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
```

(continues on next page)

(continued from previous page)

```
Configuring flash size...
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.0 seconds (effective 5962.8␣
↪kbit/s)...
Hash of data verified.
Compressed 26096 bytes to 15408...
Writing at 0x00001000... (100 %)
Wrote 26096 bytes (15408 compressed) at 0x00001000 in 0.4 seconds (effective 546.7␣
↪kbit/s)...
Hash of data verified.
Compressed 147104 bytes to 77364...
Writing at 0x00010000... (20 %)
Writing at 0x00014000... (40 %)
Writing at 0x00018000... (60 %)
Writing at 0x0001c000... (80 %)
Writing at 0x00020000... (100 %)
Wrote 147104 bytes (77364 compressed) at 0x00010000 in 1.9 seconds (effective 615.
↪5 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done
```

If there are no issues by the end of the flash process, the board will reboot and start up the "hello_world" application.

If you'd like to use the Eclipse or VS Code IDE instead of running `idf.py`, check out Eclipse Plugin, VSCode Extension.

Monitor the Output To check if "hello_world" is indeed running, type `idf.py -p PORT monitor` (Do not forget to replace PORT with your serial port name).

This command launches the *IDF Monitor* application:

```
$ idf.py -p <PORT> monitor
Running idf_monitor in directory [...]/esp/hello_world/build
Executing "python [...]/esp-idf/tools/idf_monitor.py -b 115200 [...]/esp/hello_
↪world/build/hello_world.elf"...
--- idf_monitor on <PORT> 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
ets Jun  8 2016 00:22:57
...
```

After startup and diagnostic logs scroll up, you should see "Hello world!" printed out by the application.

```
    ...
    Hello world!
    Restarting in 10 seconds...
    This is esp32 chip with 2 CPU core(s), WiFi/BT/BLE, silicon revision 1, 2MB␣
↪external flash
Minimum free heap size: 298968 bytes
    Restarting in 9 seconds...
    Restarting in 8 seconds...
    Restarting in 7 seconds...
```

To exit IDF monitor use the shortcut `Ctrl+]`.

If IDF monitor fails shortly after the upload, or, if instead of the messages above, you see random garbage similar to what is given below, your board is likely using a 26 MHz crystal. Most development board designs use 40 MHz, so ESP-IDF uses this frequency as a default value.

```
e???)(Xn@?y.!??(?PW+)??Hn9a¯/9?!?t5??P?~?k??e?ea?5?jA
~zY??Y(1?,1 ?? e???)(Xn@?y.!Dr?zY(? jpi?|?+z5Ymvp
```

If you have such a problem, do the following:

1. Exit the monitor.
2. Go back to `menuconfig`.
3. Go to `Component config` -> `Hardware Settings` -> `Main XTAL Config` -> `Main XTAL frequency`, then change *CONFIG_XTAL_FREQ_SEL* to 26 MHz.
4. After that, `build and flash` the application again.

In the current version of ESP-IDF, main XTAL frequencies supported by ESP32 are as follows:

- 26 MHz
- 40 MHz

Note: You can combine building, flashing and monitoring into one step by running:

```
idf.py -p PORT flash monitor
```

See also:

- *IDF Monitor* for handy shortcuts and more details on using IDF monitor.
- *idf.py* for a full reference of `idf.py` commands and options.

That's all that you need to get started with ESP32!

Now you are ready to try some other examples, or go straight to developing your own applications.

Important: Some of examples do not support ESP32 because required hardware is not included in ESP32 so it cannot be supported.

If building an example, please check the README file for the `Supported Targets` table. If this is present including ESP32 target, or the table does not exist at all, the example will work on ESP32.

Additional Tips

Permission issues /dev/ttyUSB0 With some Linux distributions, you may get the `Failed to open port /dev/ttyUSB0` error message when flashing the ESP32. *This can be solved by adding the current user to the dialout group*.

Python compatibility ESP-IDF supports Python 3.7 or newer. It is recommended to upgrade your operating system to a recent version satisfying this requirement. Other options include the installation of Python from sources or the use of a Python version management system such as pyenv.

Start with Board Support Package To speed up prototyping on some development boards, you can use Board Support Packages (BSPs), which makes initialization of a particular board as easy as few function calls.

A BSP typically supports all of the hardware components provided on development board. Apart from the pinout definition and initialization functions, a BSP ships with drivers for the external components such as sensors, displays, audio codecs etc.

The BSPs are distributed via *IDF Component Manager*, so they can be found in IDF Component Registry.

Here's an example of how to add ESP-WROVER-KIT BSP to your project:

```
idf.py add-dependency esp_wrover_kit
```

More examples of BSP usage can be found in BSP examples folder.

Related Documents For advanced users who want to customize the install process:

- *Updating ESP-IDF tools on Windows*
- *Establish Serial Connection with ESP32*
- Eclipse Plugin
- VSCode Extension
- *IDF Monitor*

Updating ESP-IDF tools on Windows

Install ESP-IDF tools using a script From the Windows Command Prompt, change to the directory where ESP-IDF is installed. Then run:

```
install.bat
```

For Powershell, change to the directory where ESP-IDF is installed. Then run:

```
install.ps1
```

This will download and install the tools necessary to use ESP-IDF. If the specific version of the tool is already installed, no action will be taken. The tools are downloaded and installed into a directory specified during ESP-IDF Tools Installer process. By default, this is `C:\Users\username\.espressif`.

Add ESP-IDF tools to PATH using an export script ESP-IDF tools installer creates a Start menu shortcut for "ESP-IDF Command Prompt". This shortcut opens a Command Prompt window where all the tools are already available.

In some cases, you may want to work with ESP-IDF in a Command Prompt window which wasn't started using that shortcut. If this is the case, follow the instructions below to add ESP-IDF tools to PATH.

In the command prompt where you need to use ESP-IDF, change to the directory where ESP-IDF is installed, then execute `export.bat`:

```
cd %userprofile%\esp\esp-idf
export.bat
```

Alternatively in the Powershell where you need to use ESP-IDF, change to the directory where ESP-IDF is installed, then execute `export.ps1`:

```
cd ~/esp/esp-idf
export.ps1
```

When this is done, the tools will be available in this command prompt.

Establish Serial Connection with ESP32

This section provides guidance how to establish serial connection between ESP32 and PC.

Connect ESP32 to PC Connect the ESP32 board to the PC using the USB cable. If device driver does not install automatically, identify USB to serial converter chip on your ESP32 board (or external converter dongle), search for drivers in internet and install them.

Below is the list of USB to serial converter chips installed on most of the ESP32 boards produced by Espressif together with links to the drivers:

- CP210x: CP210x USB to UART Bridge VCP Drivers
- FTDI: FTDI Virtual COM Port Drivers

Please check the board user guide for specific USB to serial converter chip used. The drivers above are primarily for reference. Under normal circumstances, the drivers should be bundled with an operating system and automatically installed upon connecting the board to the PC.

Check port on Windows Check the list of identified COM ports in the Windows Device Manager. Disconnect ESP32 and connect it back, to verify which port disappears from the list and then shows back again.

Figures below show serial port for ESP32 DevKitC and ESP32 WROVER KIT

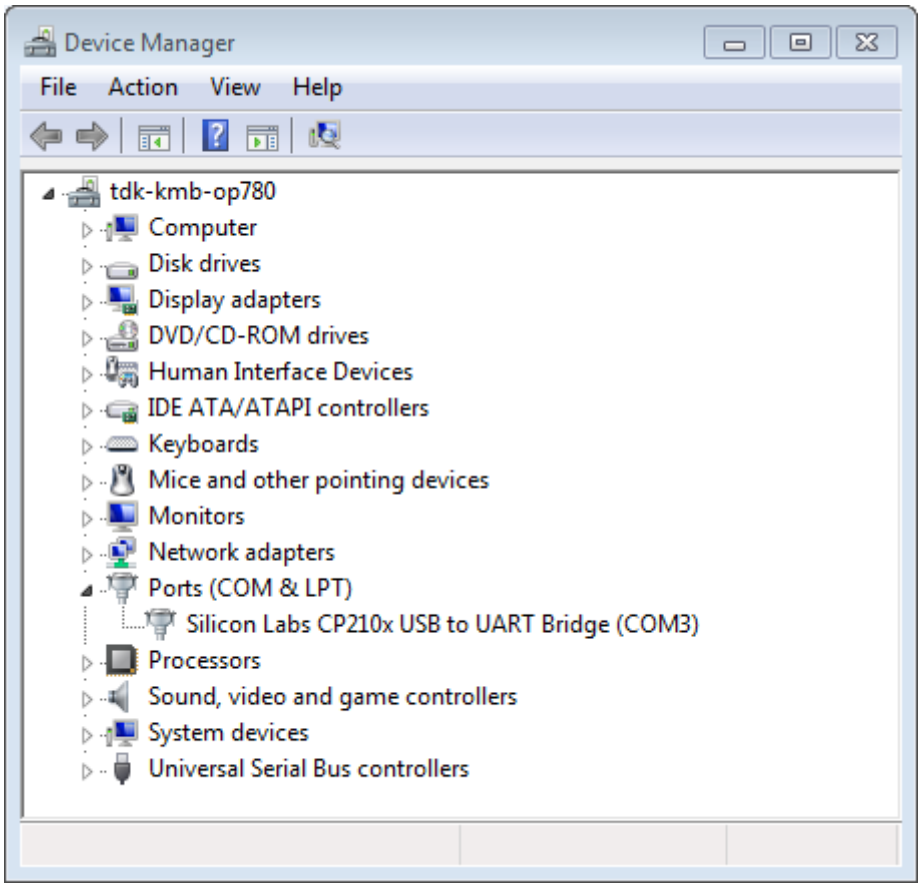


Fig. 57: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

Check port on Linux and macOS To check the device name for the serial port of your ESP32 board (or external converter dongle), run this command two times, first with the board / dongle unplugged, then with plugged in. The port which appears the second time is the one you need:

Linux

```
ls /dev/tty*
```

macOS

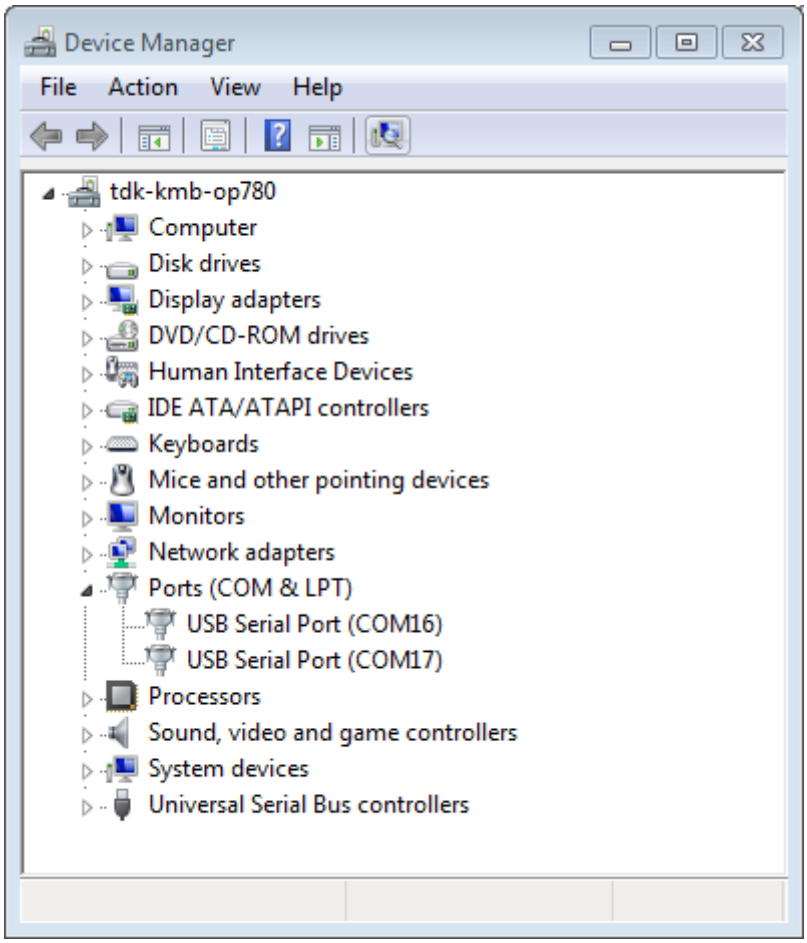


Fig. 58: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

```
ls /dev/cu.*
```

Note: macOS users: if you don't see the serial port then check you have the USB/serial drivers installed. See Section *Connect ESP32 to PC* for links to drivers. For macOS High Sierra (10.13), you may also have to explicitly allow the drivers to load. Open System Preferences -> Security & Privacy -> General and check if there is a message shown here about "System Software from developer ···" where the developer name is Silicon Labs or FTDI.

Adding user to `dialout` on Linux The currently logged user should have read and write access the serial port over USB. On most Linux distributions, this is done by adding the user to `dialout` group with the following command:

```
sudo usermod -a -G dialout $USER
```

on Arch Linux this is done by adding the user to `uucp` group with the following command:

```
sudo usermod -a -G uucp $USER
```

Make sure you re-login to enable read and write permissions for the serial port.

Verify serial connection Now verify that the serial connection is operational. You can do this using a serial terminal program by checking if you get any output on the terminal after resetting ESP32.

The default console baud rate on ESP32 is 115200.

Windows and Linux In this example we will use PuTTY SSH Client that is available for both Windows and Linux. You can use other serial programs and set communication parameters like below.

Run terminal and set identified serial port. Baud rate = 115200 (if needed, change this to the default baud rate of the chip in use), data bits = 8, stop bits = 1, and parity = N. Below are example screenshots of setting the port and such transmission parameters (in short described as 115200-8-1-N) on Windows and Linux. Remember to select exactly the same serial port you have identified in steps above.

Then open serial port in terminal and check, if you see any log printed out by ESP32. The log contents will depend on application loaded to ESP32, see *Example Output*.

Note: Close the serial terminal after verification that communication is working. If you keep the terminal session open, the serial port will be inaccessible for uploading firmware later.

macOS To spare you the trouble of installing a serial terminal program, macOS offers the **screen** command.

- As discussed in *Check port on Linux and macOS*, run:

  ```
  ls /dev/cu.*
  ```

- You should see similar output:

  ```
  /dev/cu.Bluetooth-Incoming-Port /dev/cu.SLAB_USBtoUART       /dev/cu.SLAB_
  ↪USBtoUART7
  ```

- The output will vary depending on the type and the number of boards connected to your PC. Then pick the device name of your board and run (if needed, change "115200" to the default baud rate of the chip in use):

  ```
  screen /dev/cu.device_name 115200
  ```

  Replace `device_name` with the name found running `ls /dev/cu.*`.

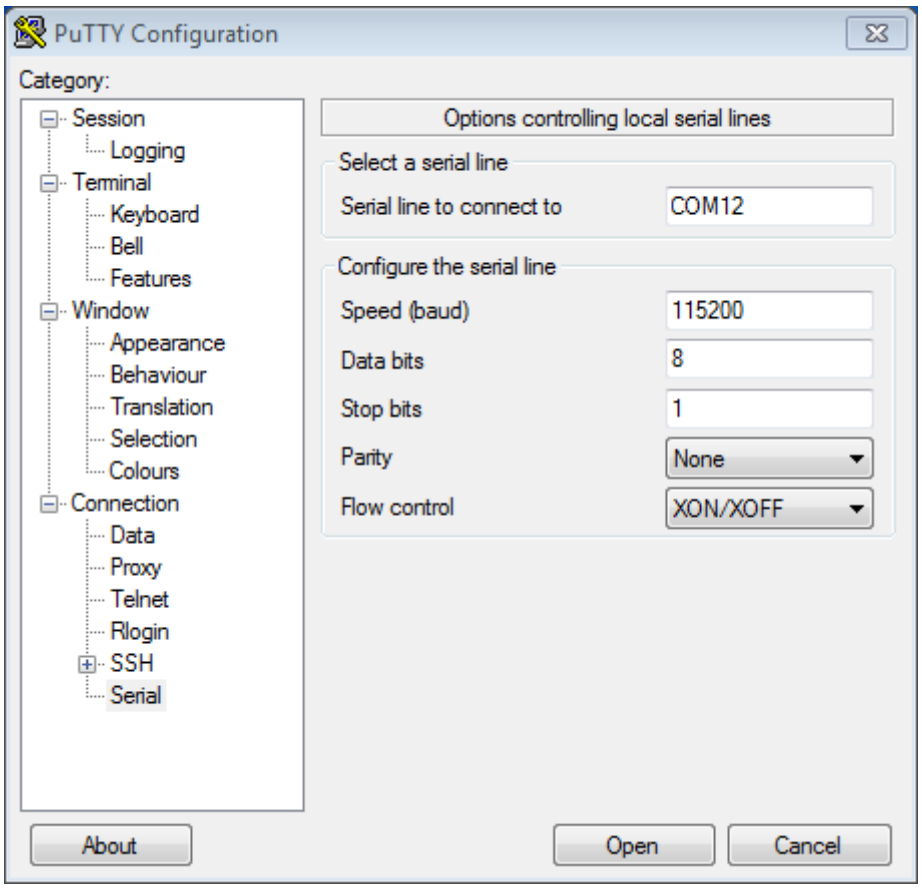


Fig. 59: Setting Serial Communication in PuTTY on Windows

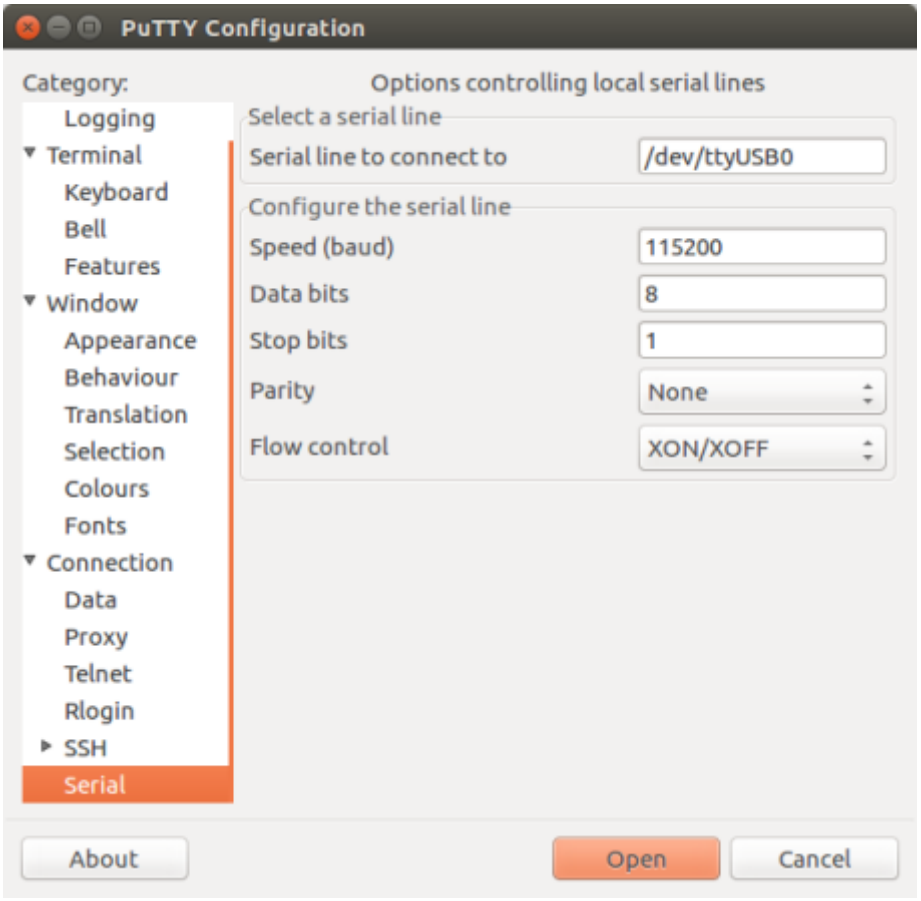


Fig. 60: Setting Serial Communication in PuTTY on Linux

- What you are looking for is some log displayed by the **screen**. The log contents will depend on application loaded to ESP32, see *Example Output*. To exit the **screen** session type Ctrl-A + \ .

Note: Do not forget to **exit the screen session** after verifying that the communication is working. If you fail to do it and just close the terminal window, the serial port will be inaccessible for uploading firmware later.

Example Output An example log is shown below. Reset the board if you do not see anything.

```
ets Jun  8 2016 00:22:57

rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0008,len:8
load:0x3fff0010,len:3464
load:0x40078000,len:7828
load:0x40080000,len:252
entry 0x40080034
I (44) boot: ESP-IDF v2.0-rc1-401-gf9fba35 2nd stage bootloader
I (45) boot: compile time 18:48:10

...
```

If you can see readable log output, it means serial connection is working and you are ready to proceed with installation and finally upload of application to ESP32.

Note: For some serial port wiring configurations, the serial RTS & DTR pins need to be disabled in the terminal program before the ESP32 will boot and produce serial output. This depends on the hardware itself, most development boards (including all Espressif boards) *do not* have this issue. The issue is present if RTS & DTR are wired directly to the EN & GPIO0 pins. See the esptool documentation for more details.

If you got here from *Step 5. First Steps on ESP-IDF* when installing s/w for ESP32 development, then you can continue with *Step 5. First Steps on ESP-IDF*.

IDF Monitor

IDF Monitor is mainly a serial terminal program which relays serial data to and from the target device ' s serial port. It also provides some IDF-specific features.

IDF Monitor can be launched from an IDF project by running `idf.py monitor`.

Keyboard Shortcuts For easy interaction with IDF Monitor, use the keyboard shortcuts given in the table.

Keyboard Shortcut	Action	Description
Ctrl+]	Exit the program	
Ctrl+T	Menu escape key	Press and follow it by one of the keys given below.
• Ctrl+T	Send the menu character itself to remote	
• Ctrl+]	Send the exit character itself to remote	
• Ctrl+P	Reset target into bootloader to pause app via RTS line	Resets the target, into bootloader via the RTS line (if connected), so that the board runs nothing. Useful when you need to wait for another device to startup.
• Ctrl+R	Reset target board via RTS	Resets the target board and re-starts the application via the RTS line (if connected).
• Ctrl+F	Build and flash the project	Pauses idf_monitor to run the project `flash` target, then resumes idf_monitor. Any changed source files are recompiled and then re-flashed. Target `encrypted-flash` is run if idf_monitor was started with argument `-E`.
• Ctrl+A (or A)	Build and flash the app only	Pauses idf_monitor to run the `app-flash` target, then resumes idf_monitor. Similar to the `flash` target, but only the main app is built and re-flashed. Target `encrypted-app-flash` is run if idf_monitor was started with argument `-E`.
• Ctrl+Y	Stop/resume log output printing on screen	Discards all incoming serial data while activated. Allows to quickly pause and examine log output without quitting the monitor.
• Ctrl+L	Stop/resume log output saved to file	Creates a file in the project directory and the output is written to that file until this is disabled with the same keyboard shortcut (or IDF Monitor exits).
• Ctrl+I (or I)	Stop/resume printing timestamps	IDF Monitor can print a timestamp in the beginning of each line. The timestamp format can be changed by the `--timestamp-format` command line argument.
• Ctrl+H (or H)	Display all keyboard shortcuts	
• Ctrl+X (or X)	Exit the program	
Ctrl+C	Interrupt running application	Pauses IDF Monitor and run GDB project debugger to debug the application at runtime. This requires :ref:CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME option to be enabled.

Any keys pressed, other than `Ctrl-]` and `Ctrl-T`, will be sent through the serial port.

IDF-specific features

Automatic Address Decoding Whenever ESP-IDF outputs a hexadecimal code address of the form `0x4_____`, IDF Monitor uses `addr2line_` to look up the location in the source code and find the function name.

If an ESP-IDF app crashes and panics, a register dump and backtrace is produced, such as the following:

```
Guru Meditation Error of type StoreProhibited occurred on core  0. Exception was␣
↪unhandled.
Register dump:
PC      : 0x400f360d  PS      : 0x00060330  A0      : 0x800dbf56  A1      :␣
↪0x3ffb7e00
A2      : 0x3ffb136c  A3      : 0x00000005  A4      : 0x00000000  A5      :␣
↪0x00000000
A6      : 0x00000000  A7      : 0x00000080  A8      : 0x00000000  A9      :␣
↪0x3ffb7dd0
A10     : 0x00000003  A11     : 0x00060f23  A12     : 0x00060f20  A13     :␣
↪0x3ffba6d0
A14     : 0x00000047  A15     : 0x0000000f  SAR     : 0x00000019  EXCCAUSE:␣
↪0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0x00000000

Backtrace: 0x400f360d:0x3ffb7e00 0x400dbf56:0x3ffb7e20 0x400dbf5e:0x3ffb7e40␣
↪0x400dbf82:0x3ffb7e60 0x400d071d:0x3ffb7e90
```

IDF Monitor adds more details to the dump:

```
Guru Meditation Error of type StoreProhibited occurred on core  0. Exception was␣
↪unhandled.
Register dump:
PC      : 0x400f360d  PS      : 0x00060330  A0      : 0x800dbf56  A1      :␣
↪0x3ffb7e00
0x400f360d: do_something_to_crash at /home/gus/esp/32/idf/examples/get-started/
↪hello_world/main/./hello_world_main.c:57
(inlined by) inner_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_
↪world/main/./hello_world_main.c:52
A2      : 0x3ffb136c  A3      : 0x00000005  A4      : 0x00000000  A5      :␣
↪0x00000000
A6      : 0x00000000  A7      : 0x00000080  A8      : 0x00000000  A9      :␣
↪0x3ffb7dd0
A10     : 0x00000003  A11     : 0x00060f23  A12     : 0x00060f20  A13     :␣
↪0x3ffba6d0
A14     : 0x00000047  A15     : 0x0000000f  SAR     : 0x00000019  EXCCAUSE:␣
↪0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0x00000000

Backtrace: 0x400f360d:0x3ffb7e00 0x400dbf56:0x3ffb7e20 0x400dbf5e:0x3ffb7e40␣
↪0x400dbf82:0x3ffb7e60 0x400d071d:0x3ffb7e90
0x400f360d: do_something_to_crash at /home/gus/esp/32/idf/examples/get-started/
↪hello_world/main/./hello_world_main.c:57
(inlined by) inner_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_
↪world/main/./hello_world_main.c:52
0x400dbf56: still_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_
↪world/main/./hello_world_main.c:47
0x400dbf5e: dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_world/
↪main/./hello_world_main.c:42
0x400dbf82: app_main at /home/gus/esp/32/idf/examples/get-started/hello_world/main/
↪./hello_world_main.c:33
0x400d071d: main_task at /home/gus/esp/32/idf/components/esp32/./cpu_start.c:254
```

To decode each address, IDF Monitor runs the following command in the background:

```
xtensa-esp32-elf-addr2line -pfiaC -e build/PROJECT.elf ADDRESS
```

Note: Set environment variable `ESP_MONITOR_DECODE` to `0` or call idf_monitor.py with specific command line

option: `idf_monitor.py --disable-address-decoding` to disable address decoding.

Target Reset on Connection By default, IDF Monitor will reset the target when connecting to it. The reset of the target chip is performed using the DTR and RTS serial lines. To prevent IDF Monitor from automatically resetting the target on connection, call IDF Monitor with the `--no-reset` option (e.g., `idf_monitor.py --no-reset`).

Note: The `--no-reset` option applies the same behavior even when connecting IDF Monitor to a particular port (e.g., `idf.py monitor --no-reset -p [PORT]`).

Launching GDB with GDBStub GDBStub is a useful runtime debugging feature that runs on the target and connects to the host over the serial port to receive debugging commands. GDBStub supports commands such as reading memory and variables, examining call stack frames etc. Although GDBStub is less versatile than JTAG debugging, it does not require any special hardware (such as a JTAG to USB bridge) as communication is done entirely over the serial port.

A target can be configured to run GDBStub in the background by setting the *CONFIG_ESP_SYSTEM_PANIC* to `GDBStub on runtime`. GDBStub will run in the background until a `Ctrl+C` message is sent over the serial port and causes the GDBStub to break (i.e., stop the execution of) the program, thus allowing GDBStub to handle debugging commands.

Furthermore, the panic handler can be configured to run GDBStub on a crash by setting the *CONFIG_ESP_SYSTEM_PANIC* to `GDBStub on panic`. When a crash occurs, GDBStub will output a special string pattern over the serial port to indicate that it is running.

In both cases (i.e., sending the `Ctrl+C` message, or receiving the special string pattern), IDF Monitor will automatically launch GDB in order to allow the user to send debugging commands. After GDB exits, the target is reset via the RTS serial line. If this line is not connected, users can reset their target (by pressing the board's Reset button).

Note: In the background, IDF Monitor runs the following command to launch GDB:

```
xtensa-esp32-elf-gdb -ex "set serial baud BAUD" -ex "target remote PORT" -ex␣
↪interrupt build/PROJECT.elf :idf_target:`Hello NAME chip`
```

Output Filtering IDF monitor can be invoked as `idf.py monitor --print-filter="xyz"`, where `--print-filter` is the parameter for output filtering. The default value is an empty string, which means that everything is printed.

Restrictions on what to print can be specified as a series of `<tag>:<log_level>` items where `<tag>` is the tag string and `<log_level>` is a character from the set `{N, E, W, I, D, V, *}` referring to a level for *logging*.

For example, `PRINT_FILTER="tag1:W"` matches and prints only the outputs written with `ESP_LOGW("tag1", ...)` or at lower verbosity level, i.e. `ESP_LOGE("tag1", ...)`. Not specifying a `<log_level>` or using `*` defaults to Verbose level.

Note: Use primary logging to disable at compilation the outputs you do not need through the *logging library*. Output filtering with IDF monitor is a secondary solution which can be useful for adjusting the filtering options without recompiling the application.

Your app tags must not contain spaces, asterisks `*`, or colons `:` to be compatible with the output filtering feature.

If the last line of the output in your app is not followed by a carriage return, the output filtering might get confused, i.e., the monitor starts to print the line and later finds out that the line should not have been written. This is a known issue and can be avoided by always adding a carriage return (especially when no output follows immediately afterwards).

Examples of Filtering Rules:

- `*` can be used to match any tags. However, the string `PRINT_FILTER="*:I tag1:E"` with regards to `tag1` prints errors only, because the rule for `tag1` has a higher priority over the rule for `*`.
- The default (empty) rule is equivalent to `*:V` because matching every tag at the Verbose level or lower means matching everything.
- `"*:N"` suppresses not only the outputs from logging functions, but also the prints made by `printf`, etc. To avoid this, use `*:E` or a higher verbosity level.
- Rules `"tag1:V"`, `"tag1:v"`, `"tag1:"`, `"tag1:*"`, and `"tag1"` are equivalent.
- Rule `"tag1:W tag1:E"` is equivalent to `"tag1:E"` because any consequent occurrence of the same tag name overwrites the previous one.
- Rule `"tag1:I tag2:W"` only prints `tag1` at the Info verbosity level or lower and `tag2` at the Warning verbosity level or lower.
- Rule `"tag1:I tag2:W tag3:N"` is essentially equivalent to the previous one because `tag3:N` specifies that `tag3` should not be printed.
- `tag3:N` in the rule `"tag1:I tag2:W tag3:N *:V"` is more meaningful because without `tag3:N` the `tag3` messages could have been printed; the errors for `tag1` and `tag2` will be printed at the specified (or lower) verbosity level and everything else will be printed by default.

A More Complex Filtering Example The following log snippet was acquired without any filtering options:

```
load:0x40078000,len:13564
entry 0x40078d4c
E (31) esp_image: image at 0x30000 has invalid magic byte
W (31) esp_image: image at 0x30000 has invalid SPI mode 255
E (39) boot: Factory app partition is not bootable
I (568) cpu_start: Pro cpu up.
I (569) heap_init: Initializing. RAM available for dynamic allocation:
I (603) cpu_start: Pro cpu start user code
D (309) light_driver: [light_init, 74]:status: 1, mode: 2
D (318) vfs: esp_vfs_register_fd_range is successful for range <54; 64) and VFS ID␣
↪1
I (328) wifi: wifi driver task: 3ffdbf84, prio:23, stack:4096, core=0
```

The captured output for the filtering options `PRINT_FILTER="wifi esp_image:E light_driver:I"` is given below:

```
E (31) esp_image: image at 0x30000 has invalid magic byte
I (328) wifi: wifi driver task: 3ffdbf84, prio:23, stack:4096, core=0
```

`The options ``PRINT_FILTER="light_driver:D esp_image:N boot:N cpu_start:N vfs:N wifi:N *:V"` show the following output:

```
load:0x40078000,len:13564
entry 0x40078d4c
I (569) heap_init: Initializing. RAM available for dynamic allocation:
D (309) light_driver: [light_init, 74]:status: 1, mode: 2
```

Known Issues with IDF Monitor

Issues Observed on Windows

- Arrow keys, as well as some other keys, do not work in GDB due to Windows Console limitations.
- Occasionally, when "idf.py" exits, it might stall for up to 30 seconds before IDF Monitor resumes.
- When "gdb" is run, it might stall for a short time before it begins communicating with the GDBStub.

Standard Toolchain Setup for Linux and macOS

Installation Step by Step This is a detailed roadmap to walk you through the installation process.

Setting up Development Environment These are the steps for setting up the ESP-IDF for your ESP32.

- *Step 1. Install Prerequisites*
- *Step 2. Get ESP-IDF*
- *Step 3. Set up the tools*
- *Step 4. Set up the environment variables*
- *Step 5. First Steps on ESP-IDF*

Step 1. Install Prerequisites In order to use ESP-IDF with the ESP32, you need to install some software packages based on your Operating System. This setup guide will help you on getting everything installed on Linux and macOS based systems.

For Linux Users To compile using ESP-IDF you will need to get the following packages. The command to run depends on which distribution of Linux you are using:

- Ubuntu and Debian:

```
sudo apt-get install git wget flex bison gperf python3 python3-pip python3-
↪venv cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0
```

- CentOS 7 & 8:

```
sudo yum -y update && sudo yum install git wget flex bison gperf python3 cmake␣
↪ninja-build ccache dfu-util libusbx
```

CentOS 7 is still supported but CentOS version 8 is recommended for a better user experience.

- Arch:

```
sudo pacman -S --needed gcc git make flex bison gperf python cmake ninja␣
↪ccache dfu-util libusb
```

Note:

- CMake version 3.16 or newer is required for use with ESP-IDF. Run "tools/idf_tools.py install cmake" to install a suitable version if your OS versions doesn't have one.
- If you do not see your Linux distribution in the above list then please check its documentation to find out which command to use for package installation.

For macOS Users ESP-IDF will use the version of Python installed by default on macOS.

- Install CMake & Ninja build:
  - If you have HomeBrew, you can run:

```
brew install cmake ninja dfu-util
```

  - If you have MacPorts, you can run:

```
sudo port install cmake ninja dfu-util
```

  - Otherwise, consult the CMake and Ninja home pages for macOS installation downloads.

- It is strongly recommended to also install ccache for faster builds. If you have HomeBrew, this can be done via `brew install ccache` or `sudo port install ccache` on MacPorts.

Note: If an error like this is shown during any step:

```
xrun: error: invalid active developer path (/Library/Developer/CommandLineTools),␣
↪missing xrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
```

Then you will need to install the XCode command line tools to continue. You can install these by running `xcode-select --install`.

Apple M1 Users If you use Apple M1 platform and see an error like this:

```
WARNING: directory for tool xtensa-esp32-elf version esp-2021r2-patch3-8.4.0 is␣
↪present, but tool was not found
ERROR: tool xtensa-esp32-elf has no installed versions. Please run 'install.sh' to␣
↪install it.
```

or:

```
zsh: bad CPU type in executable: ~/.espressif/tools/xtensa-esp32-elf/esp-2021r2-
↪patch3-8.4.0/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
```

Then you will need to install Apple Rosetta 2 by running

```
/usr/sbin/softwareupdate --install-rosetta --agree-to-license
```

Installing Python 3 Based on macOS Catalina 10.15 release notes, use of Python 2.7 is not recommended and Python 2.7 will not be included by default in future versions of macOS. Check what Python you currently have:

```
python --version
```

If the output is like `Python 2.7.17`, your default interpreter is Python 2.7. If so, also check if Python 3 isn't already installed on your computer:

```
python3 --version
```

If the above command returns an error, it means Python 3 is not installed.

Below is an overview of the steps to install Python 3.

- Installing with HomeBrew can be done as follows:

  ```
  brew install python3
  ```

- If you have MacPorts, you can run:

  ```
  sudo port install python38
  ```

Step 2. Get ESP-IDF To build applications for the ESP32, you need the software libraries provided by Espressif in ESP-IDF repository.

To get ESP-IDF, navigate to your installation directory and clone the repository with `git clone`, following instructions below specific to your operating system.

Open Terminal, and run the following commands:

```
mkdir -p ~/esp
cd ~/esp
git clone -b release/v5.0 --recursive https://github.com/espressif/esp-idf.git
```

ESP-IDF will be downloaded into `~/esp/esp-idf`.

Consult *ESP-IDF Versions* for information about which ESP-IDF version to use in a given situation.

Step 3. Set up the tools Aside from the ESP-IDF, you also need to install the tools used by ESP-IDF, such as the compiler, debugger, Python packages, etc, for projects supporting ESP32.

```
cd ~/esp/esp-idf
./install.sh esp32
```

or with Fish shell

```
cd ~/esp/esp-idf
./install.fish esp32
```

The above commands install tools for ESP32 only. If you intend to develop projects for more chip targets then you should list all of them and run for example:

```
cd ~/esp/esp-idf
./install.sh esp32,esp32s2
```

or with Fish shell

```
cd ~/esp/esp-idf
./install.fish esp32,esp32s2
```

In order to install tools for all supported targets please run the following command:

```
cd ~/esp/esp-idf
./install.sh all
```

or with Fish shell

```
cd ~/esp/esp-idf
./install.fish all
```

Note: For macOS users, if an error like this is shown during any step:

```
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: unable␣
↪to get local issuer certificate (_ssl.c:xxx)
```

You may run `Install Certificates.command` in the Python folder of your computer to install certificates. For details, see Download Error While Installing ESP-IDF Tools.

Alternative File Downloads The tools installer downloads a number of files attached to GitHub Releases. If accessing GitHub is slow then it is possible to set an environment variable to prefer Espressif's download server for GitHub asset downloads.

Note: This setting only controls individual tools downloaded from GitHub releases, it doesn't change the URLs used to access any Git repositories.

To prefer the Espressif download server when installing tools, use the following sequence of commands when running `install.sh`:

```
cd ~/esp/esp-idf
export IDF_GITHUB_ASSETS="dl.espressif.com/github_assets"
./install.sh
```

Customizing the tools installation path The scripts introduced in this step install compilation tools required by ESP-IDF inside the user home directory: `$HOME/.espressif` on Linux. If you wish to install the tools into a different directory, set the environment variable `IDF_TOOLS_PATH` before running the installation scripts. Make sure that your user account has sufficient permissions to read and write this path.

If changing the `IDF_TOOLS_PATH`, make sure it is set to the same value every time the Install script (`install.bat`, `install.ps1` or `install.sh`) and an Export script (`export.bat`, `export.ps1` or `export.sh`) are executed.

Step 4. Set up the environment variables The installed tools are not yet added to the PATH environment variable. To make the tools usable from the command line, some environment variables must be set. ESP-IDF provides another script which does that.

In the terminal where you are going to use ESP-IDF, run:

```
. $HOME/esp/esp-idf/export.sh
```

or for fish (supported only since fish version 3.0.0):

```
. $HOME/esp/esp-idf/export.fish
```

Note the space between the leading dot and the path!

If you plan to use esp-idf frequently, you can create an alias for executing `export.sh`:

1. Copy and paste the following command to your shell's profile (`.profile`, `.bashrc`, `.zprofile`, etc.)

   ```
   alias get_idf='. $HOME/esp/esp-idf/export.sh'
   ```

2. Refresh the configuration by restarting the terminal session or by running `source [path to profile]`, for example, `source ~/.bashrc`.

Now you can run `get_idf` to set up or refresh the esp-idf environment in any terminal session.

Technically, you can add `export.sh` to your shell's profile directly; however, it is not recommended. Doing so activates IDF virtual environment in every terminal session (including those where IDF is not needed), defeating the purpose of the virtual environment and likely affecting other software.

Step 5. First Steps on ESP-IDF Now since all requirements are met, the next topic will guide you on how to start your first project.

This guide will help you on the first steps using ESP-IDF. Follow this guide to start a new project on the ESP32 and build, flash, and monitor the device output.

Note: If you have not yet installed ESP-IDF, please go to *Installation* and follow the instruction in order to get all the software needed to use this guide.

Start a Project Now you are ready to prepare your application for ESP32. You can start with get-started/hello_world project from examples directory in ESP-IDF.

Important: The ESP-IDF build system does not support spaces in the paths to either ESP-IDF or to projects.

Copy the project get-started/hello_world to `~/esp` directory:

```
cd ~/esp
cp -r $IDF_PATH/examples/get-started/hello_world .
```

Note: There is a range of example projects in the examples directory in ESP-IDF. You can copy any project in the same way as presented above and run it. It is also possible to build examples in-place without copying them first.

Connect Your Device Now connect your ESP32 board to the computer and check under which serial port the board is visible.

Serial ports have the following naming patterns:

- **Linux**: starting with `/dev/tty`
- **macOS**: starting with `/dev/cu.`

If you are not sure how to check the serial port name, please refer to *Establish Serial Connection with ESP32* for full details.

Note: Keep the port name handy as you will need it in the next steps.

Configure Your Project Navigate to your `hello_world` directory, set ESP32 as the target, and run the project configuration utility `menuconfig`.

```
cd ~/esp/hello_world
idf.py set-target esp32
idf.py menuconfig
```

After opening a new project, you should first set the target with `idf.py set-target esp32`. Note that existing builds and configurations in the project, if any, will be cleared and initialized in this process. The target may be saved in the environment variable to skip this step at all. See *Select the Target Chip: set-target* for additional information.

If the previous steps have been done correctly, the following menu appears:

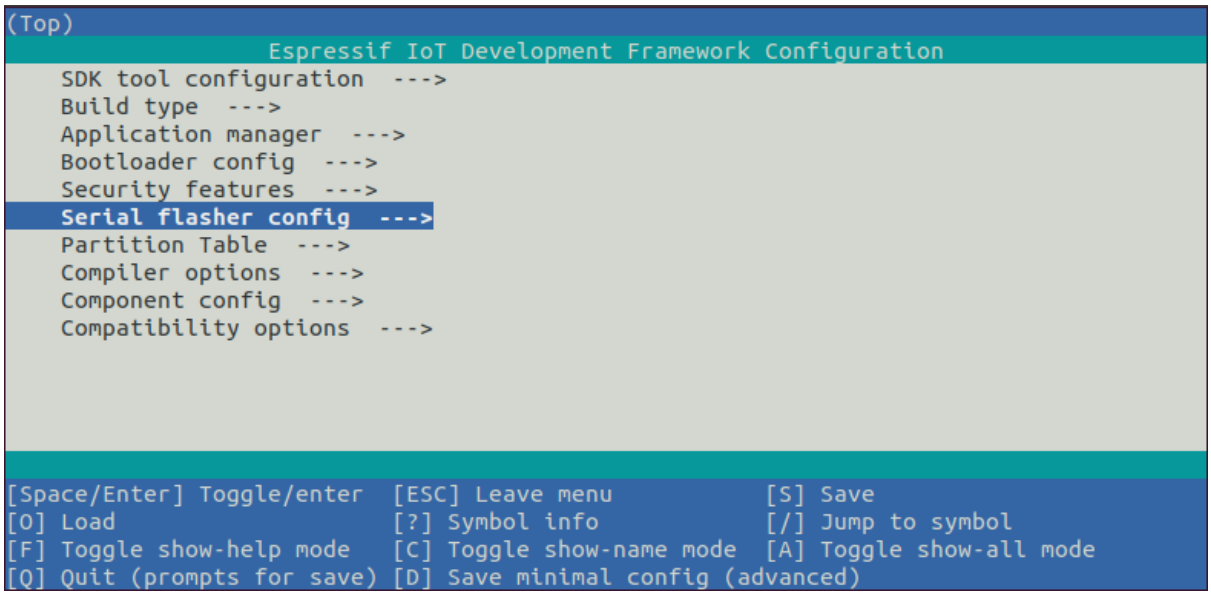


Fig. 61: Project configuration - Home window

You are using this menu to set up project specific variables, e.g., Wi-Fi network name and password, the processor speed, etc. Setting up the project with menuconfig may be skipped for "hello_world", since this example runs with

default configuration.

Attention: If you use ESP32-DevKitC board with the **ESP32-SOLO-1** module, or ESP32-DevKitM-1 board with the **ESP32-MIN1-1(1U)** module, please enable single core mode (*CONFIG_FREERTOS_UNICORE*) in menuconfig before flashing examples.

Note: The colors of the menu could be different in your terminal. You can change the appearance with the option `--style`. Please run `idf.py menuconfig --help` for further information.

If you are using one of the supported development boards, you can speed up your development by using Board Support Package. See *Additional Tips* for more information.

Build the Project Build the project by running:

```
idf.py build
```

This command will compile the application and all ESP-IDF components, then it will generate the bootloader, partition table, and application binaries.

```
$ idf.py build
Running cmake in directory /path/to/hello_world/build
Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
Warn about uninitialized values.
-- Found Git: /usr/bin/git (found version "2.17.0")
-- Building empty aws_iot component due to configuration
-- Component names: ...
-- Component paths: ...

... (more lines of build system output)

[527/527] Generating hello_world.bin
esptool.py v2.3.1

Project build complete. To flash, run this command:
../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600 write_flash -
↪-flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/hello_world.
↪bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/partition_table/
↪partition-table.bin
or run 'idf.py -p PORT flash'
```

If there are no errors, the build will finish by generating the firmware binary .bin files.

Flash onto the Device Flash the binaries that you just built (bootloader.bin, partition-table.bin and hello_world.bin) onto your ESP32 board by running:

```
idf.py -p PORT [-b BAUD] flash
```

Replace PORT with your ESP32 board' s serial port name.

You can also change the flasher baud rate by replacing BAUD with the baud rate you need. The default baud rate is `460800`.

For more information on idf.py arguments, see *idf.py*.

Note: The option `flash` automatically builds and flashes the project, so running `idf.py build` is not necessary.

Encountered Issues While Flashing? If you run the given command and see errors such as "Failed to connect", there might be several reasons for this. One of the reasons might be issues encountered by `esptool.py`, the utility that is called by the build system to reset the chip, interact with the ROM bootloader, and flash firmware. One simple solution to try is manual reset described below, and if it does not help you can find more details about possible issues in Troubleshooting.

`esptool.py` resets ESP32 automatically by asserting DTR and RTS control lines of the USB to serial converter chip, i.e., FTDI or CP210x (for more information, see *Establish Serial Connection with ESP32*). The DTR and RTS control lines are in turn connected to `GPIO0` and `CHIP_PU` (EN) pins of ESP32, thus changes in the voltage levels of DTR and RTS will boot ESP32 into Firmware Download mode. As an example, check the schematic for the ESP32 DevKitC development board.

In general, you should have no problems with the official esp-idf development boards. However, `esptool.py` is not able to reset your hardware automatically in the following cases:

- Your hardware does not have the DTR and RTS lines connected to `GPIO0` and `CHIP_PU`
- The DTR and RTS lines are configured differently
- There are no such serial control lines at all

Depending on the kind of hardware you have, it may also be possible to manually put your ESP32 board into Firmware Download mode (reset).

- For development boards produced by Espressif, this information can be found in the respective getting started guides or user guides. For example, to manually reset an ESP-IDF development board, hold down the **Boot** button (`GPIO0`) and press the **EN** button (`CHIP_PU`).
- For other types of hardware, try pulling `GPIO0` down.

Normal Operation When flashing, you will see the output log similar to the following:

```
...
esptool.py --chip esp32 -p /dev/ttyUSB0 -b 460800 --before=default_reset --
↪after=hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB␣
↪0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.bin␣
↪0x10000 hello_world.bin
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting........_
Chip is ESP32D0WDQ6 (revision 0)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 24:0a:c4:05:b9:14
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.0 seconds (effective 5962.8␣
↪kbit/s)...
Hash of data verified.
Compressed 26096 bytes to 15408...
Writing at 0x00001000... (100 %)
Wrote 26096 bytes (15408 compressed) at 0x00001000 in 0.4 seconds (effective 546.7␣
↪kbit/s)...
Hash of data verified.
Compressed 147104 bytes to 77364...
Writing at 0x00010000... (20 %)
Writing at 0x00014000... (40 %)
Writing at 0x00018000... (60 %)
Writing at 0x0001c000... (80 %)
```

(continues on next page)

(continued from previous page)

```
Writing at 0x00020000... (100 %)
Wrote 147104 bytes (77364 compressed) at 0x00010000 in 1.9 seconds (effective 615.
↪5 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done
```

If there are no issues by the end of the flash process, the board will reboot and start up the "hello_world" application.

If you'd like to use the Eclipse or VS Code IDE instead of running `idf.py`, check out Eclipse Plugin, VSCode Extension.

Monitor the Output To check if "hello_world" is indeed running, type `idf.py -p PORT monitor` (Do not forget to replace PORT with your serial port name).

This command launches the *IDF Monitor* application:

```
$ idf.py -p <PORT> monitor
Running idf_monitor in directory [...]/esp/hello_world/build
Executing "python [...]/esp-idf/tools/idf_monitor.py -b 115200 [...]/esp/hello_
↪world/build/hello_world.elf"...
--- idf_monitor on <PORT> 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
ets Jun  8 2016 00:22:57
...
```

After startup and diagnostic logs scroll up, you should see "Hello world!" printed out by the application.

```
    ...
    Hello world!
    Restarting in 10 seconds...
    This is esp32 chip with 2 CPU core(s), WiFi/BT/BLE, silicon revision 1, 2MB␣
↪external flash
Minimum free heap size: 298968 bytes
    Restarting in 9 seconds...
    Restarting in 8 seconds...
    Restarting in 7 seconds...
```

To exit IDF monitor use the shortcut `Ctrl+]`.

If IDF monitor fails shortly after the upload, or, if instead of the messages above, you see random garbage similar to what is given below, your board is likely using a 26 MHz crystal. Most development board designs use 40 MHz, so ESP-IDF uses this frequency as a default value.

```
e@@@)(Xn@@y.!@@(@PW+)@@Hn9a¯/9@!@t5@@P@~@k@@e@ea@5@jA
~zY@@Y(1@,1 @@ e@@@)(Xn@@y.!Dr@zY(@ jpi@|@+z5Ymvp
```

If you have such a problem, do the following:

1. Exit the monitor.
2. Go back to `menuconfig`.
3. Go to `Component config` -> `Hardware Settings` -> `Main XTAL Config` -> `Main XTAL frequency`, then change *CONFIG_XTAL_FREQ_SEL* to 26 MHz.
4. After that, `build and flash` the application again.

In the current version of ESP-IDF, main XTAL frequencies supported by ESP32 are as follows:

- 26 MHz
- 40 MHz

Note: You can combine building, flashing and monitoring into one step by running:

```
idf.py -p PORT flash monitor
```

See also:

- *IDF Monitor* for handy shortcuts and more details on using IDF monitor.
- *idf.py* for a full reference of `idf.py` commands and options.

That's all that you need to get started with ESP32!

Now you are ready to try some other examples, or go straight to developing your own applications.

Important: Some of examples do not support ESP32 because required hardware is not included in ESP32 so it cannot be supported.

If building an example, please check the README file for the `Supported Targets` table. If this is present including ESP32 target, or the table does not exist at all, the example will work on ESP32.

Additional Tips

Permission issues /dev/ttyUSB0 With some Linux distributions, you may get the `Failed to open port /dev/ttyUSB0` error message when flashing the ESP32. *This can be solved by adding the current user to the dialout group*.

Python compatibility ESP-IDF supports Python 3.7 or newer. It is recommended to upgrade your operating system to a recent version satisfying this requirement. Other options include the installation of Python from sources or the use of a Python version management system such as pyenv.

Start with Board Support Package To speed up prototyping on some development boards, you can use Board Support Packages (BSPs), which makes initialization of a particular board as easy as few function calls.

A BSP typically supports all of the hardware components provided on development board. Apart from the pinout definition and initialization functions, a BSP ships with drivers for the external components such as sensors, displays, audio codecs etc.

The BSPs are distributed via *IDF Component Manager*, so they can be found in IDF Component Registry.

Here's an example of how to add ESP-WROVER-KIT BSP to your project:

```
idf.py add-dependency esp_wrover_kit
```

More examples of BSP usage can be found in BSP examples folder.

Tip: Updating ESP-IDF It is recommended to update ESP-IDF from time to time, as newer versions fix bugs and/or provide new features. Please note that each ESP-IDF major and minor release version has an associated support period, and when one release branch is approaching end of life (EOL), all users are encouraged to upgrade their projects to more recent ESP-IDF releases, to find out more about support periods, see *ESP-IDF Versions*.

The simplest way to do the update is to delete the existing `esp-idf` folder and clone it again, as if performing the initial installation described in *Step 2. Get ESP-IDF*.

Another solution is to update only what has changed. *The update procedure depends on the version of ESP-IDF you are using*.

After updating ESP-IDF, execute the Install script again, in case the new ESP-IDF version requires different versions of tools. See instructions at *Step 3. Set up the tools*.

Once the new tools are installed, update the environment using the Export script. See instructions at *Step 4. Set up the environment variables*.

Related Documents

- *Establish Serial Connection with ESP32*
- Eclipse Plugin
- VSCode Extension
- *IDF Monitor*

1.4 Build Your First Project

If you already have the ESP-IDF installed and not using IDE, you can build your first project from the command line following the *Start a Project on Windows* or *Start a Project on Linux and macOS*.

1.5 Uninstall ESP-IDF

If you want to remove ESP-IDF, please follow *Uninstall ESP-IDF*.

Chapter 2

API Reference

2.1 API Conventions

This document describes conventions and assumptions common to ESP-IDF Application Programming Interfaces (APIs).

ESP-IDF provides several kinds of programming interfaces:

- C functions, structures, enums, type definitions and preprocessor macros declared in public header files of ESP-IDF components. Various pages in the API Reference section of the programming guide contain descriptions of these functions, structures and types.
- Build system functions, predefined variables and options. These are documented in the *build system guide*.
- *Kconfig* options can can be used in code and in the build system (CMakeLists.txt) files.
- *Host tools* and their command line parameters are also part of ESP-IDF interface.

ESP-IDF consists of components written specifically for ESP-IDF as well as third-party libraries. In some cases, an ESP-IDF-specific wrapper is added to the third-party library, providing an interface that is either simpler or better integrated with the rest of ESP-IDF facilities. In other cases, the original API of the third-party library is presented to the application developers.

Following sections explain some of the aspects of ESP-IDF APIs and their usage.

2.1.1 Error handling

Most ESP-IDF APIs return error codes defined with `esp_err_t` type. See *Error Handling* section for more information about error handling approaches. *Error Code Reference* contains the list of error codes returned by ESP-IDF components.

2.1.2 Configuration structures

Important: Correct initialization of configuration structures is an important part in making the application compatible with future versions of ESP-IDF.

Most initialization or configuration functions in ESP-IDF take as an argument a pointer to a configuration structure. For example:

```
const esp_timer_create_args_t my_timer_args = {
    .callback = &my_timer_callback,
    .arg = callback_arg,
    .name = "my_timer"
};
esp_timer_handle_t my_timer;
esp_err_t err = esp_timer_create(&my_timer_args, &my_timer);
```

Initialization functions never store the pointer to the configuration structure, so it is safe to allocate the structure on the stack.

The application must initialize all fields of the structure. The following is incorrect:

```
esp_timer_create_args_t my_timer_args;
my_timer_args.callback = &my_timer_callback;
/* Incorrect! Fields .arg and .name are not initialized */
esp_timer_create(&my_timer_args, &my_timer);
```

Most ESP-IDF examples use C99 designated initializers for structure initialization, since they provide a concise way of setting a subset of fields, and zero-initializing the remaining fields:

```
const esp_timer_create_args_t my_timer_args = {
    .callback = &my_timer_callback,
    /* Correct, fields .arg and .name are zero-initialized */
};
```

C++ language doesn't support the designated initializers syntax until C++20, however GCC compiler partially supports it as an extension. When using ESP-IDF APIs in C++ code, you may consider using the following pattern:

```
esp_timer_create_args_t my_timer_args = {};
/* All the fields are zero-initialized */
my_timer_args.callback = &my_timer_callback;
```

Default initializers

For some configuration structures, ESP-IDF provides macros for setting default values of fields:

```
httpd_config_t config = HTTPD_DEFAULT_CONFIG();
/* HTTPD_DEFAULT_CONFIG expands to a designated initializer.
   Now all fields are set to the default values.
   Any field can still be modified: */
config.server_port = 8081;
httpd_handle_t server;
esp_err_t err = httpd_start(&server, &config);
```

It is recommended to use default initializer macros whenever they are provided for a particular configuration structure.

2.1.3 Private APIs

Certain header files in ESP-IDF contain APIs intended to be used only in ESP-IDF source code, and not by the applications. Such header files often contain `private` or `esp_private` in their name or path. Certain components, such as *hal* only contain private APIs.

Private APIs may be removed or changed in an incompatible way between minor or patch releases.

2.1.4 Components in example projects

ESP-IDF examples contain a variety of projects demonstrating usage of ESP-IDF APIs. In order to reduce code duplication in the examples, a few common helpers are defined inside components that are used by multiple examples.

This includes components located in common_components directory, as well as some of the components located in the examples themselves. These components are not considered to be part of the ESP-IDF API.

It is not recommended to reference these components directly in custom projects (via `EXTRA_COMPONENT_DIRS` build system variable), as they may change significantly between ESP-IDF versions. When starting a new project based on an ESP-IDF example, copy both the project and the common components it depends on out of ESP-IDF, and treat the common components as part of the project. Note that the common components are written with examples in mind, and might not include all the error handling required for production applications. Take time to read the code and understand if it applicable to your use case.

2.1.5 API Stability

ESP-IDF uses Semantic Versioning as explained in the *versions page*.

Minor and bugfix releases of ESP-IDF guarantee compatibility with previous releases. The sections below explain different aspects and limitations to compatibility.

Source level compatibility

ESP-IDF guarantees source level compatibility of C functions, structures, enums, type definitions and preprocessor macros declared in public header files of ESP-IDF components. Source level compatibility implies that the application can be recompiled with the newer version of ESP-IDF without changes.

The following changes are allowed between minor versions and do not break source level compatibility:

- Deprecating functions (using the `deprecated` attribute) and header files (using a preprocessor `#warning`). Deprecations are listed in ESP-IDF relese notes. It is recommended to update the source code to use the newer functions or files that replace the deprecated ones, however this is not mandatory. Deprecated functions and files can be removed in major versions of ESP-IDF.
- Renaming components, moving source and header files between components —provided that the build system ensures that correct files are still found.
- Renaming Kconfig options. Kconfig system *renaming mechanism* ensures that the original Kconfig option names can still be used by the application in `sdkconfig` file, CMake files and source code.

Lack of binary compatibility

ESP-IDF does not guarantee binary compatibility between releases. This means that if a precompiled library is built with one ESP-IDF version, it is not guaranteed to work the same way with the next minor or bugfix release. The following are the possible changes that keep source level compatibility but not binary compatibility:

- Changing numerical values for C enum members.
- Adding new structure members or changing the order of members. See *Configuration structures* for tips that help ensure compatibility.
- Replacing an `extern` function with a `static inline` one with the same signature, or vice versa.
- Replacing a function-like macro with a compatible C function.

Other exceptions from compatibility

While we try to make upgrading to a new ESP-IDF version easy, there are parts of ESP-IDF that may change between minor versions in an incompatible way. We appreciate issue reports about any unintended breaking changes that don't fall into the categories below.

- *Private APIs*.
- *Components in example projects*.
- Features clearly marked as "beta", "preview", or "experimental".
- Changes made to mitigate security issues or to replace insecure default behaviors with a secure ones.

- Features which were never functional. For example, if it was never possible to use a certain function or an enumeration value, it may get renamed (as part of fixing it) or removed. This includes software features which depend on non-functional chip hardware features.
- Unexpected or undefined behavior (for example, due to missing validation of argument ranges) that is not documented explicitly may be fixed/changed.
- Location of *Kconfig* options in menuconfig.
- Location and names of example projects.

2.2 Application Protocols

2.2.1 ASIO port

Asio is a cross-platform C++ library, see https://think-async.com/Asio/. It provides a consistent asynchronous model using a modern C++ approach.

The ESP-IDF component *ASIO* has been moved from ESP-IDF since version v5.0 to a separate repository:

- ASIO component on GitHub

To add ASIO component in your project, please run *idf.py add-dependency espressif/asio*

Hosted Documentation

The documentation can be found on the link below:

- ASIO documentation (English)

2.2.2 ESP-Modbus

The Espressif ESP-Modbus Library (esp-modbus) supports Modbus communication in the networks based on RS485, Wi-Fi, Ethernet interfaces. The ESP-IDF component *freemodbus* has been moved from ESP-IDF since version v5.0 to a separate repository:

- ESP-Modbus component on GitHub

Hosted Documentation

The documentation can be found on the link below:

- ESP-Modbus documentation (English)

Application Example

The examples below demonstrate the ESP-Modbus library of serial, TCP ports for slave and master implementations accordingly.

- protocols/modbus/serial/mb_slave
- protocols/modbus/serial/mb_master
- protocols/modbus/tcp/mb_tcp_slave
- protocols/modbus/tcp/mb_tcp_master

Please refer to the specific example README.md for details.

Protocol References

- `https://modbus.org/specs.php`: Modbus Organization with protocol specifications.

2.2.3 ESP-MQTT

Overview

ESP-MQTT is an implementation of [MQTT](mqtt.org) protocol client (MQTT is a lightweight publish/subscribe messaging protocol).

Features

- Supports MQTT over TCP, SSL with mbedtls, MQTT over Websocket, MQTT over Websocket Secure.
- Easy to setup with URI
- Multiple instances (Multiple clients in one application)
- Support subscribing, publishing, authentication, last will messages, keep alive pings and all 3 QoS levels (it should be a fully functional client).

Application Example

- protocols/mqtt/tcp: MQTT over tcp, default port 1883
- protocols/mqtt/ssl: MQTT over tls, default port 8883
- protocols/mqtt/ssl_ds: MQTT over tls using digital signature peripheral for authentication, default port 8883.
- protocols/mqtt/ssl_mutual_auth: MQTT over tls using certificates for authentication, default port 8883
- protocols/mqtt/ssl_psk: MQTT over tls using pre-shared keys for authentication, default port 8883.
- protocols/mqtt/ws: MQTT over Websocket, default port 80
- protocols/mqtt/wss: MQTT over Websocket Secure, default port 443

Configuration

The configuration is made by setting fields in `esp_mqtt_client_config_t` struct. The configuration struct has the following sub structs to configure different aspects of the client operation.

- `broker` - Allow to set address and security verification.
- `credentials` - Client credentials for authentication.
- `session` - Configuration for MQTT session aspects.
- `network` - Networking related configuration.
- `task` - Allow to configure FreeRTOS task.
- `buffer` - Buffer size for input and output.

In the following session the most common aspects are detailed.

Broker

Address Broker address can be set by usage of `broker.address` struct. The configuration can be made by usage of `uri` field or the combination of `hostname`, `transport` and `port`. Optionally, *path* could be set, this field is useful in websocket connections.

The `uri` field is used in the following format `scheme://hostname:port/path`. - Curently support `mqtt`, `mqtts`, `ws`, `wss` schemes - MQTT over TCP samples:

- `mqtt://mqtt.eclipseprojects.io`: MQTT over TCP, default port 1883:
- `mqtt://mqtt.eclipseprojects.io:1884` MQTT over TCP, port 1884:

- `mqtt://username:password@mqtt.eclipseprojects.io:1884` MQTT over TCP, port 1884, with username and password
- MQTT over SSL samples:
  - `mqtts://mqtt.eclipseprojects.io`: MQTT over SSL, port 8883
  - `mqtts://mqtt.eclipseprojects.io:8884`: MQTT over SSL, port 8884
- MQTT over Websocket samples:
  - `ws://mqtt.eclipseprojects.io:80/mqtt`
- MQTT over Websocket Secure samples:
  - `wss://mqtt.eclipseprojects.io:443/mqtt`
- Minimal configurations:

```
const esp_mqtt_client_config_t mqtt_cfg = {
    .broker.address.uri = "mqtt://mqtt.eclipseprojects.io",
};
esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler,␣
↪client);
esp_mqtt_client_start(client);
```

- Note: By default mqtt client uses event loop library to post related mqtt events (connected, subscribed, published, etc.)

Verification For secure connections TLS is used, and to guarantee Broker's identity the `broker.verification` struct must be set. The broker certificate may be set in PEM or DER format. To select DER the equivalent `_len` field must be set, otherwise a NULL terminated string in PEM format should be provided to `certificate` field.

- Get certificate from server, example: `mqtt.eclipseprojects.io openssl s_client -showcerts -connect mqtt.eclipseprojects.io:8883 </dev/null 2>/dev/null|openssl x509 -outform PEM >mqtt_eclipse_org.pem`
- Check the sample application: `examples/mqtt_ssl`
- Configuration:

```
const esp_mqtt_client_config_t mqtt_cfg = {
    .broker = {
      .address.uri = "mqtts://mqtt.eclipseprojects.io:8883",
      .verification.certificate = (const char *)mqtt_eclipse_org_pem_start,
    },
};
```

To details on other fields check the Reference API and *TLS Server verification*.

Client Credentials All client related credentials are under the `credentials` field.

- `username`: pointer to the username used for connecting to the broker, can also be set by URI.
- `client_id`: pointer to the client id, defaults to `ESP32_%CHIPID%` where %CHIPID% are the last 3 bytes of MAC address in hex format

Authentication It's possible to set authentication parameters through the `authentication` field. The client supports the following authentication methods:

- Using a password by setting `authentication.password`.
- Muthual authentication with TLS by setting `authentication.certificate` and `authentication.key`, both can be provided in PEM or DER format.
- Using secure element available in ESP32-WROOM-32SE, setting `authentication.use_secure_element`.
- Using Digital Signature Peripheral available in some Espressif devices, setting `authentication.ds_data`.

Session For MQTT session related configurations `section` fields should be used.

Last Will and Testament MQTT allows for a last will and testament (LWT) message to notify other clients when a client ungracefully disconnects. This is configured by the following fields in the `esp_mqtt_client_config_t.session.last_will`-struct.

- `topic`: pointer to the LWT message topic
- `msg`: pointer to the LWT message
- `msg_len`: length of the LWT message, required if `msg` is not null-terminated
- `qos`: quality of service for the LWT message
- `retain`: specifies the retain flag of the LWT message

Change settings in Project Configuration Menu The settings for MQTT can be found using `idf.py menuconfig`, under Component config -> ESP-MQTT Configuration

The following settings are available:

- *CONFIG_MQTT_PROTOCOL_311*: Enables 3.1.1 version of MQTT protocol
- *CONFIG_MQTT_TRANSPORT_SSL*, *CONFIG_MQTT_TRANSPORT_WEBSOCKET*: Enables specific MQTT transport layer, such as SSL, WEBSOCKET, WEBSOCKET_SECURE
- *CONFIG_MQTT_CUSTOM_OUTBOX*: Disables default implementation of mqtt_outbox, so a specific implementaion can be supplied

Events

The following events may be posted by the MQTT client:

- `MQTT_EVENT_BEFORE_CONNECT`: The client is initialized and about to start connecting to the broker.
- `MQTT_EVENT_CONNECTED`: The client has successfully established a connection to the broker. The client is now ready to send and receive data.
- `MQTT_EVENT_DISCONNECTED`: The client has aborted the connection due to being unable to read or write data, e.g. because the server is unavailable.
- `MQTT_EVENT_SUBSCRIBED`: The broker has acknowledged the client's subscribe request. The event data will contain the message ID of the subscribe message.
- `MQTT_EVENT_UNSUBSCRIBED`: The broker has acknowledged the client's unsubscribe request. The event data will contain the message ID of the unsubscribe message.
- `MQTT_EVENT_PUBLISHED`: The broker has acknowledged the client's publish message. This will only be posted for Quality of Service level 1 and 2, as level 0 does not use acknowledgements. The event data will contain the message ID of the publish message.
- `MQTT_EVENT_DATA`: The client has received a publish message. The event data contains: message ID, name of the topic it was published to, received data and its length. For data that exceeds the internal buffer multiple *MQTT_EVENT_DATA* will be posted and *current_data_offset* and *total_data_len* from event data updated to keep track of the fragmented message.
- `MQTT_EVENT_ERROR`: The client has encountered an error. *esp_mqtt_error_type_t* from *error_handle* in the event data can be used to further determine the type of the error. The type of error will determine which parts of the *error_handle* struct is filled.

API Reference

Header File

- components/mqtt/esp-mqtt/include/mqtt_client.h

Functions

esp_mqtt_client_handle_t **esp_mqtt_client_init** (const *esp_mqtt_client_config_t* *config)

Creates *MQTT* client handle based on the configuration.

Parameters **config** – *MQTT* configuration structure
Returns mqtt_client_handle if successfully created, NULL on error

esp_err_t **esp_mqtt_client_set_uri** (*esp_mqtt_client_handle_t* client, const char *uri)

Sets *MQTT* connection URI. This API is usually used to overrides the URI configured in esp_mqtt_client_init.

Parameters
- **client** – *MQTT* client handle
- **uri** –

Returns ESP_FAIL if URI parse error, ESP_OK on success

esp_err_t **esp_mqtt_client_start** (*esp_mqtt_client_handle_t* client)

Starts *MQTT* client with already created client handle.

Parameters **client** – *MQTT* client handle
Returns ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL on other error

esp_err_t **esp_mqtt_client_reconnect** (*esp_mqtt_client_handle_t* client)

This api is typically used to force reconnection upon a specific event.

Parameters **client** – *MQTT* client handle
Returns ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL if client is in invalid state

esp_err_t **esp_mqtt_client_disconnect** (*esp_mqtt_client_handle_t* client)

This api is typically used to force disconnection from the broker.

Parameters **client** – *MQTT* client handle
Returns ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_mqtt_client_stop** (*esp_mqtt_client_handle_t* client)

Stops *MQTT* client tasks.

- Notes:
- Cannot be called from the *MQTT* event handler

Parameters **client** – *MQTT* client handle
Returns ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL if client is in invalid state

int **esp_mqtt_client_subscribe** (*esp_mqtt_client_handle_t* client, const char *topic, int qos)

Subscribe the client to defined topic with defined qos.

Notes:

- Client must be connected to send subscribe message
- This API is could be executed from a user task or from a *MQTT* event callback i.e. internal *MQTT* task (API is protected by internal mutex, so it might block if a longer data receive operation is in progress.

Parameters
- **client** – *MQTT* client handle
- **topic** –
- **qos** – // TODO describe parameters

Returns message_id of the subscribe message on success -1 on failure

int **esp_mqtt_client_unsubscribe** (*esp_mqtt_client_handle_t* client, const char *topic)

Unsubscribe the client from defined topic.

Notes:

- Client must be connected to send unsubscribe message

- It is thread safe, please refer to `esp_mqtt_client_subscribe` for details

Parameters
- **client** –*MQTT* client handle
- **topic** –

Returns message_id of the subscribe message on success -1 on failure

int **esp_mqtt_client_publish**(*esp_mqtt_client_handle_t* client, const char *topic, const char *data, int len, int qos, int retain)

Client to send a publish message to the broker.

Notes:

- This API might block for several seconds, either due to network timeout (10s) or if publishing payloads longer than internal buffer (due to message fragmentation)
- Client doesn't have to be connected for this API to work, enqueueing the messages with qos>1 (returning -1 for all the qos=0 messages if disconnected). If MQTT_SKIP_PUBLISH_IF_DISCONNECTED is enabled, this API will not attempt to publish when the client is not connected and will always return -1.
- It is thread safe, please refer to `esp_mqtt_client_subscribe` for details

Parameters
- **client** –*MQTT* client handle
- **topic** –topic string
- **data** –payload string (set to NULL, sending empty payload message)
- **len** –data length, if set to 0, length is calculated from payload string
- **qos** –QoS of publish message
- **retain** –retain flag

Returns message_id of the publish message (for QoS 0 message_id will always be zero) on success. -1 on failure.

int **esp_mqtt_client_enqueue**(*esp_mqtt_client_handle_t* client, const char *topic, const char *data, int len, int qos, int retain, bool store)

Enqueue a message to the outbox, to be sent later. Typically used for messages with qos>0, but could be also used for qos=0 messages if store=true.

This API generates and stores the publish message into the internal outbox and the actual sending to the network is performed in the mqtt-task context (in contrast to the esp_mqtt_client_publish() which sends the publish message immediately in the user task's context). Thus, it could be used as a non blocking version of esp_mqtt_client_publish().

Parameters
- **client** –*MQTT* client handle
- **topic** –topic string
- **data** –payload string (set to NULL, sending empty payload message)
- **len** –data length, if set to 0, length is calculated from payload string
- **qos** –QoS of publish message
- **retain** –retain flag
- **store** –if true, all messages are enqueued; otherwise only QoS 1 and QoS 2 are enqueued

Returns message_id if queued successfully, -1 otherwise

esp_err_t **esp_mqtt_client_destroy**(*esp_mqtt_client_handle_t* client)

Destroys the client handle.

Notes:

- Cannot be called from the *MQTT* event handler

Parameters **client** –*MQTT* client handle
Returns ESP_OK ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_mqtt_set_config**(*esp_mqtt_client_handle_t* client, const *esp_mqtt_client_config_t* *config)

Set configuration structure, typically used when updating the config (i.e. on "before_connect" event.

Parameters
- **client** –*MQTT* client handle
- **config** –*MQTT* configuration structure

Returns ESP_ERR_NO_MEM if failed to allocate ESP_ERR_INVALID_ARG if conflicts on transport configuration. ESP_OK on success

esp_err_t **esp_mqtt_client_register_event**(*esp_mqtt_client_handle_t* client, *esp_mqtt_event_id_t* event, *esp_event_handler_t* event_handler, void *event_handler_arg)

Registers *MQTT* event.

Parameters
- **client** –*MQTT* client handle
- **event** –event type
- **event_handler** –handler callback
- **event_handler_arg** –handlers context

Returns ESP_ERR_NO_MEM if failed to allocate ESP_ERR_INVALID_ARG on wrong initialization ESP_OK on success

int **esp_mqtt_client_get_outbox_size**(*esp_mqtt_client_handle_t* client)

Get outbox size.

Parameters **client** –*MQTT* client handle
Returns outbox size 0 on wrong initialization

Structures

struct **esp_mqtt_error_codes**

MQTT error code structure to be passed as a contextual information into ERROR event

Important: This structure extends `esp_tls_last_error` error structure and is backward compatible with it (so might be down-casted and treated as `esp_tls_last_error` error, but recommended to update applications if used this way previously)

Use this structure directly checking error_type first and then appropriate error code depending on the source of the error:

| error_type | related member variables | note | | MQTT_ERROR_TYPE_TCP_TRANSPORT | esp_tls_last_esp_err, esp_tls_stack_err, esp_tls_cert_verify_flags, sock_errno | Error reported from tcp_transport/esp-tls | | MQTT_ERROR_TYPE_CONNECTION_REFUSED | connect_return_code | Internal error reported from *MQTT* broker on connection |

Public Members

esp_err_t **esp_tls_last_esp_err**

last esp_err code reported from esp-tls component

int **esp_tls_stack_err**

tls specific error code reported from underlying tls stack

int **esp_tls_cert_verify_flags**

tls flags reported from underlying tls stack during certificate verification

esp_mqtt_error_type_t **error_type**

error type referring to the source of the error

esp_mqtt_connect_return_code_t **connect_return_code**

connection refused error code reported from MQTT* broker on connection

int **esp_transport_sock_errno**

errno from the underlying socket

struct **esp_mqtt_event_t**

MQTT event configuration structure

Public Members

esp_mqtt_event_id_t **event_id**

MQTT event type

esp_mqtt_client_handle_t **client**

MQTT client handle for this event

char ***data**

Data associated with this event

int **data_len**

Length of the data for this event

int **total_data_len**

Total length of the data (longer data are supplied with multiple events)

int **current_data_offset**

Actual offset for the data associated with this event

char ***topic**

Topic associated with this event

int **topic_len**

Length of the topic for this event associated with this event

int **msg_id**

MQTT messaged id of message

int **session_present**

MQTT session_present flag for connection event

esp_mqtt_error_codes_t ***error_handle**

esp-mqtt error handle including esp-tls errors as well as internal *MQTT* errors

bool **retain**

Retained flag of the message associated with this event

int **qos**

QoS of the messages associated with this event

bool **dup**

dup flag of the message associated with this event

esp_mqtt_protocol_ver_t **protocol_ver**

MQTT protocol version used for connection, defaults to value from menuconfig

struct **esp_mqtt_client_config_t**

MQTT client configuration structure

- Default values can be set via menuconfig
- All certificates and key data could be passed in PEM or DER format. PEM format must have a terminating NULL character and the related len field set to 0. DER format requires a related len field set to the correct length.

Public Members

struct *esp_mqtt_client_config_t*::*broker_t* **broker**

Broker address and security verification

struct *esp_mqtt_client_config_t*::*credentials_t* **credentials**

User credentials for broker

struct *esp_mqtt_client_config_t*::*session_t* **session**

MQTT session configuration.

struct *esp_mqtt_client_config_t*::*network_t* **network**

Network configuration

struct *esp_mqtt_client_config_t*::*task_t* **task**

FreeRTOS task configuration.

struct *esp_mqtt_client_config_t*::*buffer_t* **buffer**

Buffer size configuration.

struct **broker_t**

Broker related configuration

Public Members

struct *esp_mqtt_client_config_t*::*broker_t*::*address_t* **address**

Broker address configuration

struct *esp_mqtt_client_config_t*::*broker_t*::*verification_t* **verification**

Security verification of the broker

struct **address_t**

Broker address

- uri have precedence over other fields
- If uri isn't set at least hostname, transport and port should.

Public Members

const char ***uri**

Complete *MQTT* broker URI

const char ***hostname**

Hostname, to set ipv4 pass it as string)

esp_mqtt_transport_t **transport**

Selects transport

const char ***path**

Path in the URI

uint32_t **port**

MQTT server port

struct **verification_t**

Broker identity verification

If fields are not set broker's identity isn't verified. it's recommended to set the options in this struct for security reasons.

Public Members

bool **use_global_ca_store**

Use a global ca_store, look esp-tls documentation for details.

esp_err_t (***crt_bundle_attach**)(void *conf)

Pointer to ESP x509 Certificate Bundle attach function for the usage of certificate bundles.

const char ***certificate**

Certificate data, default is NULL, not required to verify the server.

size_t **certificate_len**

Length of the buffer pointed to by certificate.

const struct *psk_key_hint* ***psk_hint_key**

Pointer to PSK struct defined in esp_tls.h to enable PSK authentication (as alternative to certificate verification). PSK is enabled only if there are no other ways to verify broker.

bool **skip_cert_common_name_check**

Skip any validation of server certificate CN field, this reduces the security of TLS and makes the *MQTT* client susceptible to MITM attacks

const char ****alpn_protos**

NULL-terminated list of supported application protocols to be used for ALPN

struct **buffer_t**

Client buffer size configuration

Client have two buffers for input and output respectivelly.

Public Members

int **size**

size of *MQTT* send/receive buffer

int **out_size**

size of *MQTT* output buffer. If not defined, defaults to the size defined by `buffer_size`

struct **credentials_t**

Client related credentials for authentication.

Public Members

const char ***username**

MQTT username

const char ***client_id**

Set *MQTT* client identifier. Ignored if set_null_client_id == true If NULL set the default client id. Default client id is `ESP32_CHIPID%` where `CHIPID%` are last 3 bytes of MAC address in hex format

bool **set_null_client_id**

Selects a NULL client id

struct *esp_mqtt_client_config_t*::*credentials_t*::*authentication_t* **authentication**

Client authentication

struct **authentication_t**

Client authentication

Fields related to client authentication by broker

For mutual authentication using TLS, user could select certificate and key, secure element or digital signature peripheral if available.

Public Members

const char ***password**

MQTT password

const char ***certificate**

Certificate for ssl mutual authentication, not required if mutual authentication is not needed. Must be provided with `key`.

size_t **certificate_len**

Length of the buffer pointed to by certificate.

const char ***key**

Private key for SSL mutual authentication, not required if mutual authentication is not needed. If it is not NULL, also `certificate` has to be provided.

size_t **key_len**

Length of the buffer pointed to by key.

const char ***key_password**

Client key decryption password, not PEM nor DER, if provided `key_password_len` must be correctly set.

int **key_password_len**

Length of the password pointed to by `key_password`

bool **use_secure_element**

Enable secure element, available in ESP32-ROOM-32SE, for SSL connection

void ***ds_data**

Carrier of handle for digital signature parameters, digital signature peripheral is available in some Espressif devices.

struct **network_t**

Network related configuration

Public Members

int **reconnect_timeout_ms**

Reconnect to the broker after this value in miliseconds if auto reconnect is not disabled (defaults to 10s)

int **timeout_ms**

Abort network operation if it is not completed after this value, in milliseconds (defaults to 10s).

int **refresh_connection_after_ms**

Refresh connection after this value (in milliseconds)

bool **disable_auto_reconnect**

Client will reconnect to server (when errors/disconnect). Set `disable_auto_reconnect=true` to disable

struct **session_t**

MQTT Session related configuration

Public Members

struct *esp_mqtt_client_config_t*::*session_t*::*last_will_t* **last_will**

Last will configuration

bool **disable_clean_session**

MQTT clean session, default clean_session is true

int **keepalive**

MQTT keepalive, default is 120 seconds

bool **disable_keepalive**

Set `disable_keepalive=true` to turn off keep-alive mechanism, keepalive is active by default. Note: setting the config value `keepalive` to `0` doesn't disable keepalive feature, but uses a default keepalive period

esp_mqtt_protocol_ver_t **protocol_ver**

MQTT protocol version used for connection.

int **message_retransmit_timeout**

timeout for retransmitting of failed packet

struct **last_will_t**

Last Will and Testament message configuration.

Public Members

const char ***topic**

LWT (Last Will and Testament) message topic

const char ***msg**

LWT message, may be NULL terminated

int **msg_len**

LWT message length, if msg isn't NULL terminated must have the correct length

int **qos**

LWT message QoS

int **retain**

LWT retained message flag

struct **task_t**

Client task configuration

Public Members

int **priority**

MQTT task priority

int **stack_size**

MQTT task stack size

Macros

MQTT_ERROR_TYPE_ESP_TLS

MQTT_ERROR_TYPE_TCP_TRANSPORT error type hold all sorts of transport layer errors, including ESP-TLS error, but in the past only the errors from MQTT_ERROR_TYPE_ESP_TLS layer were reported, so the ESP-TLS error type is re-defined here for backward compatibility

Type Definitions

typedef struct esp_mqtt_client ***esp_mqtt_client_handle_t**

typedef enum *esp_mqtt_event_id_t* **esp_mqtt_event_id_t**

MQTT event types.

User event handler receives context data in `esp_mqtt_event_t` structure with

- `client` - *MQTT* client handle
- various other data depending on event type

typedef enum *esp_mqtt_connect_return_code_t* **esp_mqtt_connect_return_code_t**

MQTT connection error codes propagated via ERROR event

typedef enum *esp_mqtt_error_type_t* **esp_mqtt_error_type_t**

MQTT connection error codes propagated via ERROR event

typedef enum *esp_mqtt_transport_t* **esp_mqtt_transport_t**

typedef enum *esp_mqtt_protocol_ver_t* **esp_mqtt_protocol_ver_t**

MQTT protocol version used for connection

typedef struct *esp_mqtt_error_codes* **esp_mqtt_error_codes_t**

MQTT error code structure to be passed as a contextual information into ERROR event

Important: This structure extends `esp_tls_last_error` error structure and is backward compatible with it (so might be down-casted and treated as `esp_tls_last_error` error, but recommended to update applications if used this way previously)

Use this structure directly checking error_type first and then appropriate error code depending on the source of the error:

| error_type | related member variables | note | | MQTT_ERROR_TYPE_TCP_TRANSPORT | esp_tls_last_esp_err, esp_tls_stack_err, esp_tls_cert_verify_flags, sock_errno | Error reported from

tcp_transport/esp-tls | | MQTT_ERROR_TYPE_CONNECTION_REFUSED | connect_return_code | Internal error reported from *MQTT* broker on connection |

typedef struct *esp_mqtt_event_t* **esp_mqtt_event_t**

MQTT event configuration structure

typedef *esp_mqtt_event_t* ***esp_mqtt_event_handle_t**

typedef *esp_err_t* (***mqtt_event_callback_t**)(*esp_mqtt_event_handle_t* event)

typedef struct *esp_mqtt_client_config_t* **esp_mqtt_client_config_t**

MQTT client configuration structure

- Default values can be set via menuconfig
- All certificates and key data could be passed in PEM or DER format. PEM format must have a terminating NULL character and the related len field set to 0. DER format requires a related len field set to the correct length.

Enumerations

enum **esp_mqtt_event_id_t**

MQTT event types.

User event handler receives context data in `esp_mqtt_event_t` structure with

- `client` - *MQTT* client handle
- various other data depending on event type

Values:

enumerator **MQTT_EVENT_ANY**

enumerator **MQTT_EVENT_ERROR**

on error event, additional context: connection return code, error handle from esp_tls (if supported)

enumerator **MQTT_EVENT_CONNECTED**

connected event, additional context: session_present flag

enumerator **MQTT_EVENT_DISCONNECTED**

disconnected event

enumerator **MQTT_EVENT_SUBSCRIBED**

subscribed event, additional context:

- msg_id message id
- data pointer to the received data
- data_len length of the data for this event

enumerator **MQTT_EVENT_UNSUBSCRIBED**

unsubscribed event

enumerator **MQTT_EVENT_PUBLISHED**

published event, additional context: msg_id

enumerator **MQTT_EVENT_DATA**

data event, additional context:

- msg_id message id
- topic pointer to the received topic
- topic_len length of the topic
- data pointer to the received data
- data_len length of the data for this event
- current_data_offset offset of the current data for this event
- total_data_len total length of the data received
- retain retain flag of the message
- qos QoS level of the message
- dup dup flag of the message Note: Multiple MQTT_EVENT_DATA could be fired for one message, if it is longer than internal buffer. In that case only first event contains topic pointer and length, other contain data only with current data length and current data offset updating.

enumerator **MQTT_EVENT_BEFORE_CONNECT**

The event occurs before connecting

enumerator **MQTT_EVENT_DELETED**

Notification on delete of one message from the internal outbox, if the message couldn't have been sent and acknowledged before expiring defined in OUTBOX_EXPIRED_TIMEOUT_MS. (events are not posted upon deletion of successfully acknowledged messages)

- This event id is posted only if MQTT_REPORT_DELETED_MESSAGES==1
- Additional context: msg_id (id of the deleted message).

enum **esp_mqtt_connect_return_code_t**

MQTT connection error codes propagated via ERROR event

Values:

enumerator **MQTT_CONNECTION_ACCEPTED**

Connection accepted

enumerator **MQTT_CONNECTION_REFUSE_PROTOCOL**

MQTT connection refused reason: Wrong protocol

enumerator **MQTT_CONNECTION_REFUSE_ID_REJECTED**

MQTT connection refused reason: ID rejected

enumerator **MQTT_CONNECTION_REFUSE_SERVER_UNAVAILABLE**

MQTT connection refused reason: Server unavailable

enumerator **MQTT_CONNECTION_REFUSE_BAD_USERNAME**

MQTT connection refused reason: Wrong user

enumerator **MQTT_CONNECTION_REFUSE_NOT_AUTHORIZED**

MQTT connection refused reason: Wrong username or password

enum **esp_mqtt_error_type_t**

MQTT connection error codes propagated via ERROR event

Values:

enumerator **MQTT_ERROR_TYPE_NONE**

enumerator **MQTT_ERROR_TYPE_TCP_TRANSPORT**

enumerator **MQTT_ERROR_TYPE_CONNECTION_REFUSED**

enum **esp_mqtt_transport_t**

Values:

enumerator **MQTT_TRANSPORT_UNKNOWN**

enumerator **MQTT_TRANSPORT_OVER_TCP**

MQTT over TCP, using scheme: `MQTT`

enumerator **MQTT_TRANSPORT_OVER_SSL**

MQTT over SSL, using scheme: `MQTTS`

enumerator **MQTT_TRANSPORT_OVER_WS**

MQTT over Websocket, using scheme:: `ws`

enumerator **MQTT_TRANSPORT_OVER_WSS**

MQTT over Websocket Secure, using scheme: `wss`

enum **esp_mqtt_protocol_ver_t**

MQTT protocol version used for connection

Values:

enumerator **MQTT_PROTOCOL_UNDEFINED**

enumerator **MQTT_PROTOCOL_V_3_1**

enumerator **MQTT_PROTOCOL_V_3_1_1**

enumerator **MQTT_PROTOCOL_V_5**

2.2.4 ESP-TLS

Overview

The ESP-TLS component provides a simplified API interface for accessing the commonly used TLS functionality. It supports common scenarios like CA certification validation, SNI, ALPN negotiation, non-blocking connection among others. All the configuration can be specified in the `esp_tls_cfg_t` data structure. Once done, TLS communication can be conducted using the following APIs:

- *esp_tls_init()*: for initializing the TLS connection handle.
- *esp_tls_conn_new_sync()*: for opening a new blocking TLS connection.
- *esp_tls_conn_new_async()*: for opening a new non-blocking TLS connection.
- *esp_tls_conn_read()*: for reading from the connection.
- *esp_tls_conn_write()*: for writing into the connection.
- *esp_tls_conn_destroy()*: for freeing up the connection.

Any application layer protocol like HTTP1, HTTP2 etc can be executed on top of this layer.

Application Example

Simple HTTPS example that uses ESP-TLS to establish a secure socket connection: protocols/https_request.

Tree structure for ESP-TLS component

```
├── esp_tls.c
├── esp_tls.h
├── esp_tls_mbedtls.c
├── esp_tls_wolfssl.c
└── private_include
    ├── esp_tls_mbedtls.h
    └── esp_tls_wolfssl.h
```

The ESP-TLS component has a file esp-tls/esp_tls.h which contain the public API headers for the component. Internally ESP-TLS component uses one of the two SSL/TLS Libraries between mbedtls and wolfssl for its operation. API specific to mbedtls are present in esp-tls/private_include/esp_tls_mbedtls.h and API specific to wolfssl are present in esp-tls/private_include/esp_tls_wolfssl.h.

TLS Server verification

The ESP-TLS provides multiple options for TLS server verification on the client side. The ESP-TLS client can verify the server by validating the peer's server certificate or with the help of pre-shared keys. The user should select only one of the following options in the *esp_tls_cfg_t* structure for TLS server verification. If no option is selected then client will return a fatal error by default at the time of the TLS connection setup.

- **cacert_buf** and **cacert_bytes**: The CA certificate can be provided in a buffer to the *esp_tls_cfg_t* structure. The ESP-TLS will use the CA certificate present in the buffer to verify the server. The following variables in *esp_tls_cfg_t* structure must be set.
  - `cacert_buf` - pointer to the buffer which contains the CA cert.
  - `cacert_bytes` - size of the CA certificate in bytes.
- **use_global_ca_store**: The `global_ca_store` can be initialized and set at once. Then it can be used to verify the server for all the ESP-TLS connections which have set `use_global_ca_store = true` in their respective *esp_tls_cfg_t* structure. See API Reference section below on information regarding different API used for initializing and setting up the `global_ca_store`.
- **crt_bundle_attach**: The ESP x509 Certificate Bundle API provides an easy way to include a bundle of custom x509 root certificates for TLS server verification. More details can be found at *ESP x509 Certificate Bundle*
- **psk_hint_key**: To use pre-shared keys for server verification, *CONFIG_ESP_TLS_PSK_VERIFICATION* should be enabled in the ESP-TLS menuconfig. Then the pointer to PSK hint and key should be provided to the *esp_tls_cfg_t* structure. The ESP-TLS will use the PSK for server verification only when no other option regarding the server verification is selected.
- **skip server verification**: This is an insecure option provided in the ESP-TLS for testing purpose. The option can be set by enabling *CONFIG_ESP_TLS_INSECURE* and *CONFIG_ESP_TLS_SKIP_SERVER_CERT_VERIFY* in the ESP-TLS menuconfig. When this option is enabled the ESP-TLS will skip server verification by default when no other options for server verification are selected in the *esp_tls_cfg_t* structure. *WARNING:Enabling this option comes with a potential risk of establishing a TLS connection with a server which has a fake identity, provided that the server certificate is not provided either through API or other mechanism like ca_store etc.*

ESP-TLS Server cert selection hook

The ESP-TLS component provides an option to set the server cert selection hook when using the mbedTLS stack. This provides an ability to configure and use a certificate selection callback during server handshake, to select a certificate to present to the client based on the TLS extensions supplied in the client hello (alpn, sni, etc). To enable this feature, please enable *CONFIG_ESP_TLS_SERVER_CERT_SELECT_HOOK* in the ESP-TLS menuconfig. The certificate selection callback can be configured in the `esp_tls_cfg_t` structure as follows:

```
int cert_selection_callback(mbedtls_ssl_context *ssl)
{
    /* Code that the callback should execute */
    return 0;
}

esp_tls_cfg_t cfg = {
    cert_select_cb = cert_section_callback,
};
```

Underlying SSL/TLS Library Options

The ESP-TLS component has an option to use mbedtls or wolfssl as their underlying SSL/TLS library. By default only mbedtls is available and is used, wolfssl SSL/TLS library is available publicly at https://github.com/espressif/esp-wolfssl. The repository provides wolfssl component in binary format, it also provides few examples which are useful for understanding the API. Please refer the repository README.md for information on licensing and other options. Please see below option for using wolfssl in your project.

Note: *As the library options are internal to ESP-TLS, switching the libraries will not change ESP-TLS specific code for a project.*

How to use wolfssl with ESP-IDF

There are two ways to use wolfssl in your project

1) Directly add wolfssl as a component in your project with following three commands.:

```
(First change directory (cd) to your project directory)
mkdir components
cd components
git clone https://github.com/espressif/esp-wolfssl.git
```

2) Add wolfssl as an extra component in your project.

- Download wolfssl with:

```
git clone https://github.com/espressif/esp-wolfssl.git
```

- Include esp-wolfssl in ESP-IDF with setting EXTRA_COMPONENT_DIRS in CMakeLists.txt of your project as done in wolfssl/examples. For reference see Optional Project variables in *build-system*.

After above steps, you will have option to choose wolfssl as underlying SSL/TLS library in configuration menu of your project as follows:

```
idf.py menuconfig -> ESP-TLS -> choose SSL/TLS Library -> mbedtls/wolfssl
```

Comparison between mbedtls and wolfssl

The following table shows a typical comparison between wolfssl and mbedtls when protocols/https_request example *(which has server authentication)* was run with both SSL/TLS libraries and with all respective configurations

set to default. *(mbedtls IN_CONTENT length and OUT_CONTENT length were set to 16384 bytes and 4096 bytes respectively)*

Property	Wolfssl	Mbedtls
Total Heap Consumed	~19 Kb	~37 Kb
Task Stack Used	~2.2 Kb	~3.6 Kb
Bin size	~858 Kb	~736 Kb

Note: *These values are subject to change with change in configuration options and version of respective libraries.*

ATECC608A (Secure Element) with ESP-TLS

ESP-TLS provides support for using ATECC608A cryptoauth chip with ESP32-WROOM-32SE. Use of ATECC608A is supported only when ESP-TLS is used with mbedTLS as its underlying SSL/TLS stack. ESP-TLS uses mbedtls as its underlying TLS/SSL stack by default unless changed manually.

Note: ATECC608A chip on ESP32-WROOM-32SE must be already configured and provisioned, for details refer esp_cryptoauth_utility

To enable the secure element support, and use it in you project for TLS connection, you will have to follow below steps

1) Add esp-cryptoauthlib in your project, for details please refer esp-cryptoauthlib with ESP_IDF
2) Enable following menuconfig option:

```
menuconfig->Component config->ESP-TLS->Use Secure Element (ATECC608A) with ESP-
↪TLS
```

3) Select type of ATECC608A chip with following option:

```
menuconfig->Component config->esp-cryptoauthlib->Choose Type of ATECC608A chip
```

to know more about different types of ATECC608A chips and how to obtain type of ATECC608A connected to your ESP module please visit ATECC608A chip type

4) Enable use of ATECC608A in ESP-TLS by providing following config option in *esp_tls_cfg_t*

```
esp_tls_cfg_t cfg = {
    /* other configurations options */
    .use_secure_element = true,
};
```

API Reference

Header File

- components/esp-tls/esp_tls.h

Functions

esp_tls_t ***esp_tls_init** (void)

Create TLS connection.

This function allocates and initializes esp-tls structure handle.

Returns tls Pointer to esp-tls as esp-tls handle if successfully initialized, NULL if allocation error

esp_tls_t ***esp_tls_conn_http_new**(const char *url, const *esp_tls_cfg_t* *cfg)

Create a new blocking TLS/SSL connection with a given "HTTP" url.

Note: This API is present for backward compatibility reasons. Alternative function with the same functionality is `esp_tls_conn_http_new_sync` (and its asynchronous version `esp_tls_conn_http_new_async`)

Parameters

- **`url`** –**[in]** url of host.
- **`cfg`** –**[in]** TLS configuration as esp_tls_cfg_t. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to 'esp_tls_cfg_t'. At a minimum, this structure should be zero-initialized.

Returns pointer to esp_tls_t, or NULL if connection couldn't be opened.

int **esp_tls_conn_new_sync**(const char *hostname, int hostlen, int port, const *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new blocking TLS/SSL connection.

This function establishes a TLS/SSL connection with the specified host in blocking manner.

Parameters

- **`hostname`** –**[in]** Hostname of the host.
- **`hostlen`** –**[in]** Length of hostname.
- **`port`** –**[in]** Port number of the host.
- **`cfg`** –**[in]** TLS configuration as esp_tls_cfg_t. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to esp_tls_cfg_t. At a minimum, this structure should be zero-initialized.
- **`tls`** –**[in]** Pointer to esp-tls as esp-tls handle.

Returns

- -1 If connection establishment fails.
- 1 If connection establishment is successful.
- 0 If connection state is in progress.

int **esp_tls_conn_http_new_sync**(const char *url, const *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new blocking TLS/SSL connection with a given "HTTP" url.

The behaviour is same as esp_tls_conn_new_sync() API. However this API accepts host's url.

Parameters

- **`url`** –**[in]** url of host.
- **`cfg`** –**[in]** TLS configuration as esp_tls_cfg_t. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to 'esp_tls_cfg_t'. At a minimum, this structure should be zero-initialized.
- **`tls`** –**[in]** Pointer to esp-tls as esp-tls handle.

Returns

- -1 If connection establishment fails.
- 1 If connection establishment is successful.
- 0 If connection state is in progress.

int **esp_tls_conn_new_async**(const char *hostname, int hostlen, int port, const *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new non-blocking TLS/SSL connection.

This function initiates a non-blocking TLS/SSL connection with the specified host, but due to its non-blocking nature, it doesn't wait for the connection to get established.

Parameters

- **`hostname`** –**[in]** Hostname of the host.
- **`hostlen`** –**[in]** Length of hostname.
- **`port`** –**[in]** Port number of the host.
- **`cfg`** –**[in]** TLS configuration as esp_tls_cfg_t. `non_block` member of this structure should be set to be true.
- **`tls`** –**[in]** pointer to esp-tls as esp-tls handle.

Returns

- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

int **esp_tls_conn_http_new_async**(const char *url, const *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new non-blocking TLS/SSL connection with a given "HTTP" url.

The behaviour is same as esp_tls_conn_new_async() API. However this API accepts host's url.

Parameters

- **url** –**[in]** url of host.
- **cfg** –**[in]** TLS configuration as esp_tls_cfg_t.
- **tls** –**[in]** pointer to esp-tls as esp-tls handle.

Returns

- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

ssize_t **esp_tls_conn_write**(*esp_tls_t* *tls, const void *data, size_t datalen)

Write from buffer 'data' into specified tls connection.

Parameters

- **tls** –**[in]** pointer to esp-tls as esp-tls handle.
- **data** –**[in]** Buffer from which data will be written.
- **datalen** –**[in]** Length of data buffer.

Returns

- >=0 if write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.
- <0 if write operation was not successful, because either an error occured or an action must be taken by the calling process.
- ESP_TLS_ERR_SSL_WANT_READ/ ESP_TLS_ERR_SSL_WANT_WRITE. if the handshake is incomplete and waiting for data to be available for reading. In this case this functions needs to be called again when the underlying transport is ready for operation.

ssize_t **esp_tls_conn_read**(*esp_tls_t* *tls, void *data, size_t datalen)

Read from specified tls connection into the buffer 'data'.

Parameters

- **tls** –**[in]** pointer to esp-tls as esp-tls handle.
- **data** –**[in]** Buffer to hold read data.
- **datalen** –**[in]** Length of data buffer.

Returns

- >0 if read operation was successful, the return value is the number of bytes actually read from the TLS/SSL connection.
- 0 if read operation was not successful. The underlying connection was closed.
- <0 if read operation was not successful, because either an error occured or an action must be taken by the calling process.

int **esp_tls_conn_destroy**(*esp_tls_t* *tls)

Close the TLS/SSL connection and free any allocated resources.

This function should be called to close each tls connection opened with esp_tls_conn_new_sync() (or esp_tls_conn_http_new_sync()) and esp_tls_conn_new_async() (or esp_tls_conn_http_new_async()) APIs.

Parameters **tls** –**[in]** pointer to esp-tls as esp-tls handle.

Returns - 0 on success

- -1 if socket error or an invalid argument

ssize_t **esp_tls_get_bytes_avail**(*esp_tls_t* *tls)

Return the number of application data bytes remaining to be read from the current record.

This API is a wrapper over mbedtls's mbedtls_ssl_get_bytes_avail() API.

Parameters **tls** –**[in]** pointer to esp-tls as esp-tls handle.
Returns
- -1 in case of invalid arg
- bytes available in the application data record read buffer

esp_err_t **esp_tls_get_conn_sockfd**(*esp_tls_t* *tls, int *sockfd)

Returns the connection socket file descriptor from esp_tls session.

Parameters
- **tls** –**[in]** handle to esp_tls context
- **sockfd** –**[out]** int pointer to sockfd value.

Returns - ESP_OK on success and value of sockfd will be updated with socket file descriptor for connection
- ESP_ERR_INVALID_ARG if (tls == NULL || sockfd == NULL)

void ***esp_tls_get_ssl_context**(*esp_tls_t* *tls)

Returns the ssl context.

Parameters **tls** –**[in]** handle to esp_tls context
Returns - ssl_ctx pointer to ssl context of underlying TLS layer on success
- NULL in case of error

esp_err_t **esp_tls_init_global_ca_store**(void)

Create a global CA store, initially empty.

This function should be called if the application wants to use the same CA store for multiple connections. This function initialises the global CA store which can be then set by calling esp_tls_set_global_ca_store(). To be effective, this function must be called before any call to esp_tls_set_global_ca_store().

Returns
- ESP_OK if creating global CA store was successful.
- ESP_ERR_NO_MEM if an error occured when allocating the mbedTLS resources.

esp_err_t **esp_tls_set_global_ca_store**(const unsigned char *cacert_pem_buf, const unsigned int cacert_pem_bytes)

Set the global CA store with the buffer provided in pem format.

This function should be called if the application wants to set the global CA store for multiple connections i.e. to add the certificates in the provided buffer to the certificate chain. This function implicitly calls esp_tls_init_global_ca_store() if it has not already been called. The application must call this function before calling esp_tls_conn_new().

Parameters
- **cacert_pem_buf** –**[in]** Buffer which has certificates in pem format. This buffer is used for creating a global CA store, which can be used by other tls connections.
- **cacert_pem_bytes** –**[in]** Length of the buffer.

Returns
- ESP_OK if adding certificates was successful.
- Other if an error occured or an action must be taken by the calling process.

void **esp_tls_free_global_ca_store**(void)

Free the global CA store currently being used.

The memory being used by the global CA store to store all the parsed certificates is freed up. The application can call this API if it no longer needs the global CA store.

esp_err_t **esp_tls_get_and_clear_last_error**(*esp_tls_error_handle_t* h, int *esp_tls_code, int *esp_tls_flags)

Returns last error in esp_tls with detailed mbedtls related error codes. The error information is cleared internally upon return.

Parameters
- **h** –**[in]** esp-tls error handle.

- **esp_tls_code** –**[out]** last error code returned from mbedtls api (set to zero if none) This pointer could be NULL if caller does not care about esp_tls_code
- **esp_tls_flags** –**[out]** last certification verification flags (set to zero if none) This pointer could be NULL if caller does not care about esp_tls_code

Returns

- ESP_ERR_INVALID_STATE if invalid parameters
- ESP_OK (0) if no error occurred
- specific error code (based on ESP_ERR_ESP_TLS_BASE) otherwise

esp_err_t **esp_tls_get_and_clear_error_type**(*esp_tls_error_handle_t* h, *esp_tls_error_type_t* err_type, int *error_code)

Returns the last error captured in esp_tls of a specific type The error information is cleared internally upon return.

Parameters

- **h** –**[in]** esp-tls error handle.
- **err_type** –**[in]** specific error type
- **error_code** –**[out]** last error code returned from mbedtls api (set to zero if none) This pointer could be NULL if caller does not care about esp_tls_code

Returns

- ESP_ERR_INVALID_STATE if invalid parameters
- ESP_OK if a valid error returned and was cleared

esp_err_t **esp_tls_get_error_handle**(*esp_tls_t* *tls, *esp_tls_error_handle_t* *error_handle)

Returns the ESP-TLS error_handle.

Parameters

- **tls** –**[in]** handle to esp_tls context
- **error_handle** –**[out]** pointer to the error handle.

Returns

- ESP_OK on success and error_handle will be updated with the ESP-TLS error handle.
- ESP_ERR_INVALID_ARG if (tls == NULL || error_handle == NULL)

mbedtls_x509_crt ***esp_tls_get_global_ca_store**(void)

Get the pointer to the global CA store currently being used.

The application must first call esp_tls_set_global_ca_store(). Then the same CA store could be used by the application for APIs other than esp_tls.

Note: Modifying the pointer might cause a failure in verifying the certificates.

Returns

- Pointer to the global CA store currently being used if successful.
- NULL if there is no global CA store set.

esp_err_t **esp_tls_plain_tcp_connect**(const char *host, int hostlen, int port, const *esp_tls_cfg_t* *cfg, *esp_tls_error_handle_t* error_handle, int *sockfd)

Creates a plain TCP connection, returning a valid socket fd on success or an error handle.

Parameters

- **host** –**[in]** Hostname of the host.
- **hostlen** –**[in]** Length of hostname.
- **port** –**[in]** Port number of the host.
- **cfg** –**[in]** ESP-TLS configuration as esp_tls_cfg_t.
- **error_handle** –**[out]** ESP-TLS error handle holding potential errors occurred during connection
- **sockfd** –**[out]** Socket descriptor if successfully connected on TCP layer

Returns ESP_OK on success ESP_ERR_INVALID_ARG if invalid output parameters ESP-TLS based error codes on failure

Structures

struct **psk_key_hint**

ESP-TLS preshared key and hint structure.

Public Members

const uint8_t ***key**

key in PSK authentication mode in binary format

const size_t **key_size**

length of the key

const char ***hint**

hint in PSK authentication mode in string format

struct **tls_keep_alive_cfg**

esp-tls client session ticket ctx

Keep alive parameters structure

Public Members

bool **keep_alive_enable**

Enable keep-alive timeout

int **keep_alive_idle**

Keep-alive idle time (second)

int **keep_alive_interval**

Keep-alive interval time (second)

int **keep_alive_count**

Keep-alive packet retry send count

struct **esp_tls_cfg**

ESP-TLS configuration parameters.

Note: Note about format of certificates:

- This structure includes certificates of a Certificate Authority, of client or server as well as private keys, which may be of PEM or DER format. In case of PEM format, the buffer must be NULL terminated (with NULL character included in certificate size).
- Certificate Authority' s certificate may be a chain of certificates in case of PEM format, but could be only one certificate in case of DER format
- Variables names of certificates and private key buffers and sizes are defined as unions providing backward compatibility for legacy *_pem_buf and *_pem_bytes names which suggested only PEM format was supported. It is encouraged to use generic names such as cacert_buf and cacert_bytes.

Public Members

const char **alpn_protos

Application protocols required for HTTP2. If HTTP2/ALPN support is required, a list of protocols that should be negotiated. The format is length followed by protocol name. For the most common cases the following is ok: const char **alpn_protos = { “h2”, NULL };

- where ‘h2’ is the protocol name

const unsigned char *cacert_buf

Certificate Authority’s certificate in a buffer. Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char *cacert_pem_buf

CA certificate buffer legacy name

unsigned int cacert_bytes

Size of Certificate Authority certificate pointed to by cacert_buf (including NULL-terminator in case of PEM format)

unsigned int cacert_pem_bytes

Size of Certificate Authority certificate legacy name

const unsigned char *clientcert_buf

Client certificate in a buffer Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char *clientcert_pem_buf

Client certificate legacy name

unsigned int clientcert_bytes

Size of client certificate pointed to by clientcert_pem_buf (including NULL-terminator in case of PEM format)

unsigned int clientcert_pem_bytes

Size of client certificate legacy name

const unsigned char *clientkey_buf

Client key in a buffer Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char *clientkey_pem_buf

Client key legacy name

unsigned int clientkey_bytes

Size of client key pointed to by clientkey_pem_buf (including NULL-terminator in case of PEM format)

unsigned int clientkey_pem_bytes

Size of client key legacy name

const unsigned char ***clientkey_password**

Client key decryption password string

unsigned int **clientkey_password_len**

String length of the password pointed to by clientkey_password

bool **non_block**

Configure non-blocking mode. If set to true the underneath socket will be configured in non blocking mode after tls session is established

bool **use_secure_element**

Enable this option to use secure element or atecc608a chip (Integrated with ESP32-WROOM-32SE)

int **timeout_ms**

Network timeout in milliseconds. Note: If this value is not set, by default the timeout is set to 10 seconds. If you wish that the session should wait indefinitely then please use a larger value e.g., INT32_MAX

bool **use_global_ca_store**

Use a global ca_store for all the connections in which this bool is set.

const char ***common_name**

If non-NULL, server certificate CN must match this name. If NULL, server certificate CN must match hostname.

bool **skip_common_name**

Skip any validation of server certificate CN field

tls_keep_alive_cfg_t ***keep_alive_cfg**

Enable TCP keep-alive timeout for SSL connection

const *psk_hint_key_t* ***psk_hint_key**

Pointer to PSK hint and key. if not NULL (and certificates are NULL) then PSK authentication is enabled with configured setup. Important note: the pointer must be valid for connection

esp_err_t (***crt_bundle_attach**)(void *conf)

Function pointer to esp_crt_bundle_attach. Enables the use of certification bundle for server verification, must be enabled in menuconfig

void ***ds_data**

Pointer for digital signature peripheral context

bool **is_plain_tcp**

Use non-TLS connection: When set to true, the esp-tls uses plain TCP transport rather then TLS/SSL connection. Note, that it is possible to connect using a plain tcp transport directly with esp_tls_plain_tcp_connect() API

struct ifreq ***if_name**

The name of interface for data to go through. Use the default interface without setting

esp_tls_addr_family_t **addr_family**

The address family to use when connecting to a host.

Type Definitions

typedef enum *esp_tls_conn_state* **esp_tls_conn_state_t**

ESP-TLS Connection State.

typedef enum *esp_tls_role* **esp_tls_role_t**

typedef struct *psk_key_hint* **psk_hint_key_t**

ESP-TLS preshared key and hint structure.

typedef struct *tls_keep_alive_cfg* **tls_keep_alive_cfg_t**

esp-tls client session ticket ctx

Keep alive parameters structure

typedef enum *esp_tls_addr_family* **esp_tls_addr_family_t**

typedef struct *esp_tls_cfg* **esp_tls_cfg_t**

ESP-TLS configuration parameters.

Note: Note about format of certificates:

- This structure includes certificates of a Certificate Authority, of client or server as well as private keys, which may be of PEM or DER format. In case of PEM format, the buffer must be NULL terminated (with NULL character included in certificate size).
- Certificate Authority's certificate may be a chain of certificates in case of PEM format, but could be only one certificate in case of DER format
- Variables names of certificates and private key buffers and sizes are defined as unions providing backward compatibility for legacy *_pem_buf and *_pem_bytes names which suggested only PEM format was supported. It is encouraged to use generic names such as cacert_buf and cacert_bytes.

---

typedef struct esp_tls **esp_tls_t**

Enumerations

enum **esp_tls_conn_state**

ESP-TLS Connection State.

Values:

enumerator **ESP_TLS_INIT**

enumerator **ESP_TLS_CONNECTING**

enumerator **ESP_TLS_HANDSHAKE**

enumerator **ESP_TLS_FAIL**

enumerator **ESP_TLS_DONE**

enum **esp_tls_role**

Values:

enumerator **ESP_TLS_CLIENT**

enumerator **ESP_TLS_SERVER**

enum **esp_tls_addr_family**

Values:

enumerator **ESP_TLS_AF_UNSPEC**

Unspecified address family.

enumerator **ESP_TLS_AF_INET**

IPv4 address family.

enumerator **ESP_TLS_AF_INET6**

IPv6 address family.

Header File

- components/esp-tls/esp_tls_errors.h

Structures

struct **esp_tls_last_error**

Error structure containing relevant errors in case tls error occurred.

Public Members

esp_err_t **last_error**

error code (based on ESP_ERR_ESP_TLS_BASE) of the last occurred error

int **esp_tls_error_code**

esp_tls error code from last esp_tls failed api

int **esp_tls_flags**

last certification verification flags

Macros

ESP_ERR_ESP_TLS_BASE

Starting number of ESP-TLS error codes

ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME

Error if hostname couldn't be resolved upon tls connection

ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET

Failed to create socket

ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY

Unsupported protocol family

ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST

Failed to connect to host

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED

failed to set/get socket option

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT

new connection in esp_tls_low_level_conn connection timeouted

ESP_ERR_ESP_TLS_SE_FAILED

ESP_ERR_ESP_TLS_TCP_CLOSED_FIN

ESP_ERR_MBEDTLS_CERT_PARTLY_OK

mbedtls parse certificates was partly successful

ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SETUP_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_WRITE_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED

mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED

mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED

mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED

mbedtls api returned failed

ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED

wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED

wolfSSL api returned error

ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED

wolfSSL api returned error

ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED

wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED

wolfSSL api returned failed

ESP_ERR_WOLFSSL_CTX_SETUP_FAILED

wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_SETUP_FAILED

wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_WRITE_FAILED

wolfSSL api returned failed

ESP_TLS_ERR_SSL_WANT_READ

Definition of errors reported from IO API (potentially non-blocking) in case of error:

- esp_tls_conn_read()
- esp_tls_conn_write()

ESP_TLS_ERR_SSL_WANT_WRITE

ESP_TLS_ERR_SSL_TIMEOUT

Type Definitions

typedef struct *esp_tls_last_error* ***esp_tls_error_handle_t**

typedef struct *esp_tls_last_error* **esp_tls_last_error_t**

Error structure containing relevant errors in case tls error occurred.

Enumerations

enum **esp_tls_error_type_t**

Definition of different types/sources of error codes reported from different components

Values:

enumerator **ESP_TLS_ERR_TYPE_UNKNOWN**

enumerator **ESP_TLS_ERR_TYPE_SYSTEM**

System error — errno

enumerator **ESP_TLS_ERR_TYPE_MBEDTLS**

Error code from mbedTLS library

enumerator **ESP_TLS_ERR_TYPE_MBEDTLS_CERT_FLAGS**

Certificate flags defined in mbedTLS

enumerator **ESP_TLS_ERR_TYPE_ESP**

ESP-IDF error type — esp_err_t

enumerator **ESP_TLS_ERR_TYPE_WOLFSSL**

Error code from wolfSSL library

enumerator **ESP_TLS_ERR_TYPE_WOLFSSL_CERT_FLAGS**

Certificate flags defined in wolfSSL

enumerator **ESP_TLS_ERR_TYPE_MAX**

Last err type — invalid entry

2.2.5 ESP HTTP Client

Overview

`esp_http_client` provides an API for making HTTP/S requests from ESP-IDF applications. The steps to use this API are as follows:

- *`esp_http_client_init()`*: Creates an *`esp_http_client_config_t`* instance i.e. a HTTP client handle based on the given *`esp_http_client_config_t`* configuration. This function must be the first to be called; default values will be assumed for the configuration values that are not explicitly defined by the user.
- *`esp_http_client_perform()`*: Performs all operations of the esp_http_client - opening the connection, exchanging data and closing the connection (as required), while blocking the current task until its completion. All related events will be invoked through the event handler (as specified in *`esp_http_client_config_t`*).
- *`esp_http_client_cleanup()`*: Closes the connection (if any) and frees up all the memory allocated to the HTTP client instance. This must be the last function to be called after the completion of operations.

Application Example

Simple example that uses ESP HTTP Client to make HTTP/S requests at protocols/esp_http_client.

Basic HTTP request

Check out the example functions `http_rest_with_url` and `http_rest_with_hostname_path` in the application example for implementation details.

Persistent Connections

Persistent connection means that the HTTP client can re-use the same connection for several exchanges. If the server does not request to close the connection with the `Connection: close` header, the connection is not dropped but is instead kept open and used for further requests.

To allow ESP HTTP client to take full advantage of persistent connections, one should make as many requests as possible using the same handle instance.

Check out the example functions `http_rest_with_url` and `http_rest_with_hostname_path` in the application example. Here, once the connection is created, multiple requests (`GET`, `POST`, `PUT`, etc.) are made before the connection is closed.

HTTPS Request

ESP HTTP client supports SSL connections using **mbedTLS**, with the `url` configuration starting with `https` scheme or `transport_type` set to `HTTP_TRANSPORT_OVER_SSL`. HTTPS support can be configured via *CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS* (enabled by default).

Note: While making HTTPS requests, if server verification is needed, additional root certificate (in PEM format) needs to be provided to the `cert_pem` member in `esp_http_client_config_t` configuration. Users can also use the `ESP x509 Certificate Bundle` for server verification using the `crt_bundle_attach` member of the `esp_http_client_config_t` configuration.

Check out the example functions `https_with_url` and `https_with_hostname_path` in the application example. (Implementation details of the above note are found here)

HTTP Stream

Some applications need to open the connection and control the exchange of data actively (data streaming). In such cases, the application flow is different from regular requests. Example flow is given below:

- *esp_http_client_init()*: Create a HTTP client handle
- `esp_http_client_set_*` or `esp_http_client_delete_*`: Modify the HTTP connection parameters (optional)
- *esp_http_client_open()*: Open the HTTP connection with `write_len` parameter (content length that needs to be written to server), set `write_len=0` for read-only connection
- *esp_http_client_write()*: Write data to server with a maximum length equal to `write_len` of *esp_http_client_open()* function; no need to call this function for `write_len=0`
- *esp_http_client_fetch_headers()*: Read the HTTP Server response headers, after sending the request headers and server data (if any). Returns the `content-length` from the server and can be succeeded by *esp_http_client_get_status_code()* for getting the HTTP status of the connection.
- *esp_http_client_read()*: Read the HTTP stream
- *esp_http_client_close()*: Close the connection
- *esp_http_client_cleanup()*: Release allocated resources

Check out the example function `http_perform_as_stream_reader` in the application example for implementation details.

HTTP Authentication

ESP HTTP client supports both Basic and Digest Authentication.

- Users can provide the username and password in the `url` or the `username` and `password` members of the `esp_http_client_config_t` configuration. For `auth_type = HTTP_AUTH_TYPE_BASIC`, the HTTP client takes only 1 perform operation to pass the authentication process.

- If `auth_type = HTTP_AUTH_TYPE_NONE`, but the `username` and `password` fields are present in the configuration, the HTTP client takes 2 perform operations. The client will receive the `401 Unauthorized` header in its first attempt to connect to the server. Based on this information, it decides which authentication method to choose and performs it in the second operation.
- Check out the example functions `http_auth_basic`, `http_auth_basic_redirect` (for Basic authentication) and `http_auth_digest` (for Digest authentication) in the application example for implementation details.

Examples of Authentication Configuration

- Authentication with URI

```
esp_http_client_config_t config = {
    .url = "http://user:passwd@httpbin.org/basic-auth/user/passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

- Authentication with username and password entry

```
esp_http_client_config_t config = {
    .url = "http://httpbin.org/basic-auth/user/passwd",
    .username = "user",
    .password = "passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

API Reference

Header File

- components/esp_http_client/include/esp_http_client.h

Functions

esp_http_client_handle_t **esp_http_client_init** (const *esp_http_client_config_t* *config)

Start a HTTP session This function must be the first function to call, and it returns a esp_http_client_handle_t that you must use as input to other functions in the interface. This call MUST have a corresponding call to esp_http_client_cleanup when the operation is complete.

Parameters **config** –**[in]** The configurations, see `http_client_config_t`

Returns

- `esp_http_client_handle_t`
- NULL if any errors

esp_err_t **esp_http_client_perform** (*esp_http_client_handle_t* client)

Invoke this function after `esp_http_client_init` and all the options calls are made, and will perform the transfer as described in the options. It must be called with the same esp_http_client_handle_t as input as the esp_http_client_init call returned. esp_http_client_perform performs the entire request in either blocking or non-blocking manner. By default, the API performs request in a blocking manner and returns when done, or if it failed, and in non-blocking manner, it returns if EAGAIN/EWOULDBLOCK or EINPROGRESS is encountered, or if it failed. And in case of non-blocking request, the user may call this API multiple times unless request & response is complete or there is a failure. To enable non-blocking esp_http_client_perform(), `is_async` member of *esp_http_client_config_t* must be set while making a call to esp_http_client_init() API. You can do any amount of calls to esp_http_client_perform while using the same esp_http_client_handle_t. The underlying connection may be kept open if the server allows it. If you intend to transfer more than one file, you are even encouraged to do so. esp_http_client will then attempt to re-use the same connection for the following transfers, thus making the operations faster, less CPU intense and using less network resources. Just note that you will have to use `esp_http_client_set_**` between the invokes to set options for the following esp_http_client_perform.

Note: You must never call this function simultaneously from two places using the same client handle. Let the function return first before invoking it another time. If you want parallel transfers, you must use several esp_http_client_handle_t. This function include `esp_http_client_open` -> `esp_http_client_write` -> `esp_http_client_fetch_headers` -> `esp_http_client_read` (and option) `esp_http_client_close`.

Parameters **`client`** –The esp_http_client handle

Returns

- ESP_OK on successful
- ESP_FAIL on error

esp_err_t **`esp_http_client_set_url`**(*esp_http_client_handle_t* client, const char *url)

Set URL for client, when performing this behavior, the options in the URL will replace the old ones.

Parameters

- **`client`** –**[in]** The esp_http_client handle
- **`url`** –**[in]** The url

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **`esp_http_client_set_post_field`**(*esp_http_client_handle_t* client, const char *data, int len)

Set post data, this function must be called before `esp_http_client_perform`. Note: The data parameter passed to this function is a pointer and this function will not copy the data.

Parameters

- **`client`** –**[in]** The esp_http_client handle
- **`data`** –**[in]** post data pointer
- **`len`** –**[in]** post length

Returns

- ESP_OK
- ESP_FAIL

int **`esp_http_client_get_post_field`**(*esp_http_client_handle_t* client, char **data)

Get current post field information.

Parameters

- **`client`** –**[in]** The client
- **`data`** –**[out]** Point to post data pointer

Returns Size of post data

esp_err_t **`esp_http_client_set_header`**(*esp_http_client_handle_t* client, const char *key, const char *value)

Set http request header, this function must be called after esp_http_client_init and before any perform function.

Parameters

- **`client`** –**[in]** The esp_http_client handle
- **`key`** –**[in]** The header key
- **`value`** –**[in]** The header value

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **`esp_http_client_get_header`**(*esp_http_client_handle_t* client, const char *key, char **value)

Get http request header. The value parameter will be set to NULL if there is no header which is same as the key specified, otherwise the address of header value will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Parameters

- **client** –**[in]** The esp_http_client handle
- **key** –**[in]** The header key
- **value** –**[out]** The header value

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_http_client_get_username**(*esp_http_client_handle_t* client, char **value)

Get http request username. The address of username buffer will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Parameters

- **client** –**[in]** The esp_http_client handle
- **value** –**[out]** The username value

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_set_username**(*esp_http_client_handle_t* client, const char *username)

Set http request username. The value of username parameter will be assigned to username buffer. If the username parameter is NULL then username buffer will be freed.

Parameters

- **client** –**[in]** The esp_http_client handle
- **username** –**[in]** The username value

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_get_password**(*esp_http_client_handle_t* client, char **value)

Get http request password. The address of password buffer will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Parameters

- **client** –**[in]** The esp_http_client handle
- **value** –**[out]** The password value

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_set_password**(*esp_http_client_handle_t* client, const char *password)

Set http request password. The value of password parameter will be assigned to password buffer. If the password parameter is NULL then password buffer will be freed.

Parameters

- **client** –**[in]** The esp_http_client handle
- **password** –**[in]** The password value

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_set_authtype**(*esp_http_client_handle_t* client, *esp_http_client_auth_type_t* auth_type)

Set http request auth_type.

Parameters

- **client** –**[in]** The esp_http_client handle
- **auth_type** –**[in]** The esp_http_client auth type

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

int **esp_http_client_get_errno**(*esp_http_client_handle_t* client)

Get HTTP client session errno.

Parameters **client** –**[in]** The esp_http_client handle

Returns

- (-1) if invalid argument
- errno

esp_err_t **esp_http_client_set_method**(*esp_http_client_handle_t* client, *esp_http_client_method_t* method)

Set http request method.

Parameters

- **client** –**[in]** The esp_http_client handle
- **method** –**[in]** The method

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_set_timeout_ms**(*esp_http_client_handle_t* client, int timeout_ms)

Set http request timeout.

Parameters

- **client** –**[in]** The esp_http_client handle
- **timeout_ms** –**[in]** The timeout value

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

esp_err_t **esp_http_client_delete_header**(*esp_http_client_handle_t* client, const char *key)

Delete http request header.

Parameters

- **client** –**[in]** The esp_http_client handle
- **key** –**[in]** The key

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_http_client_open**(*esp_http_client_handle_t* client, int write_len)

This function will be open the connection, write all header strings and return.

Parameters

- **client** –**[in]** The esp_http_client handle
- **write_len** –**[in]** HTTP Content length need to write to the server

Returns

- ESP_OK
- ESP_FAIL

int **esp_http_client_write**(*esp_http_client_handle_t* client, const char *buffer, int len)

This function will write data to the HTTP connection previously opened by esp_http_client_open()

Parameters

- **client** –**[in]** The esp_http_client handle
- **buffer** –The buffer
- **len** –**[in]** This value must not be larger than the write_len parameter provided to esp_http_client_open()

Returns

- (-1) if any errors
- Length of data written

int64_t **esp_http_client_fetch_headers** (*esp_http_client_handle_t* client)

This function need to call after esp_http_client_open, it will read from http stream, process all receive headers.

Parameters **client** –**[in]** The esp_http_client handle
Returns
- (0) if stream doesn't contain content-length header, or chunked encoding (checked by `esp_http_client_is_chunked` response)
- (-1: ESP_FAIL) if any errors
- (-ESP_ERR_HTTP_EAGAIN = -0x7007) if call is timed-out before any data was ready
- Download data length defined by content-length header

bool **esp_http_client_is_chunked_response** (*esp_http_client_handle_t* client)

Check response data is chunked.

Parameters **client** –**[in]** The esp_http_client handle
Returns true or false

int **esp_http_client_read** (*esp_http_client_handle_t* client, char *buffer, int len)

Read data from http stream.

Note: (-ESP_ERR_HTTP_EAGAIN = -0x7007) is returned when call is timed-out before any data was ready

---

Parameters
- **client** –**[in]** The esp_http_client handle
- **buffer** –The buffer
- **len** –**[in]** The length

Returns
- (-1) if any errors
- Length of data was read

int **esp_http_client_get_status_code** (*esp_http_client_handle_t* client)

Get http response status code, the valid value if this function invoke after `esp_http_client_perform`

Parameters **client** –**[in]** The esp_http_client handle
Returns Status code

int64_t **esp_http_client_get_content_length** (*esp_http_client_handle_t* client)

Get http response content length (from header Content-Length) the valid value if this function invoke after `esp_http_client_perform`

Parameters **client** –**[in]** The esp_http_client handle
Returns
- (-1) Chunked transfer
- Content-Length value as bytes

esp_err_t **esp_http_client_close** (*esp_http_client_handle_t* client)

Close http connection, still kept all http request resources.

Parameters **client** –**[in]** The esp_http_client handle
Returns
- ESP_OK
- ESP_FAIL

esp_err_t **esp_http_client_cleanup** (*esp_http_client_handle_t* client)

This function must be the last function to call for an session. It is the opposite of the esp_http_client_init function and must be called with the same handle as input that a esp_http_client_init call returned. This might close all connections this handle has used and possibly has kept open until now. Don't call this function if you intend to transfer more files, re-using handles is a key to good performance with esp_http_client.

Parameters **client** –**[in]** The esp_http_client handle

Returns
- ESP_OK
- ESP_FAIL

esp_http_client_transport_t **esp_http_client_get_transport_type** (*esp_http_client_handle_t* client)

Get transport type.

Parameters **client** –**[in]** The esp_http_client handle
Returns
- HTTP_TRANSPORT_UNKNOWN
- HTTP_TRANSPORT_OVER_TCP
- HTTP_TRANSPORT_OVER_SSL

esp_err_t **esp_http_client_set_redirection** (*esp_http_client_handle_t* client)

Set redirection URL. When received the 30x code from the server, the client stores the redirect URL provided by the server. This function will set the current URL to redirect to enable client to execute the redirection request. When `disable_auto_redirect` is set, the client will not call this function but the event `HTTP_EVENT_REDIRECT` will be dispatched giving the user contol over the redirection event.

Parameters **client** –**[in]** The esp_http_client handle
Returns
- ESP_OK
- ESP_FAIL

void **esp_http_client_add_auth** (*esp_http_client_handle_t* client)

On receiving HTTP Status code 401, this API can be invoked to add authorization information.

Note: There is a possibility of receiving body message with redirection status codes, thus make sure to flush off body data after calling this API.

Parameters **client** –**[in]** The esp_http_client handle

bool **esp_http_client_is_complete_data_received** (*esp_http_client_handle_t* client)

Checks if entire data in the response has been read without any error.

Parameters **client** –**[in]** The esp_http_client handle
Returns
- true
- false

int **esp_http_client_read_response** (*esp_http_client_handle_t* client, char *buffer, int len)

Helper API to read larger data chunks This is a helper API which internally calls `esp_http_client_read` multiple times till the end of data is reached or till the buffer gets full.

Parameters
- **client** –**[in]** The esp_http_client handle
- **buffer** –The buffer
- **len** –**[in]** The buffer length

Returns
- Length of data was read

esp_err_t **esp_http_client_flush_response** (*esp_http_client_handle_t* client, int *len)

Process all remaining response data This uses an internal buffer to repeatedly receive, parse, and discard response data until complete data is processed. As no additional user-supplied buffer is required, this may be preferrable to `esp_http_client_read_response` in situations where the content of the response may be ignored.

Parameters
- **client** –**[in]** The esp_http_client handle
- **len** –Length of data discarded

Returns

- ESP_OK If successful, len will have discarded length
- ESP_FAIL If failed to read response
- ESP_ERR_INVALID_ARG If the client is NULL

esp_err_t **esp_http_client_get_url**(*esp_http_client_handle_t* client, char *url, const int len)

Get URL from client.

Parameters

- **client** –**[in]** The esp_http_client handle
- **url** –**[inout]** The buffer to store URL
- **len** –**[in]** The buffer length

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_http_client_get_chunk_length**(*esp_http_client_handle_t* client, int *len)

Get Chunk-Length from client.

Parameters

- **client** –**[in]** The esp_http_client handle
- **len** –**[out]** Variable to store length

Returns

- ESP_OK If successful, len will have length of current chunk
- ESP_FAIL If the server is not a chunked server
- ESP_ERR_INVALID_ARG If the client or len are NULL

Structures

struct **esp_http_client_event**

HTTP Client events data.

Public Members

esp_http_client_event_id_t **event_id**

event_id, to know the cause of the event

esp_http_client_handle_t **client**

esp_http_client_handle_t context

void ***data**

data of the event

int **data_len**

data length of data

void ***user_data**

user_data context, from *esp_http_client_config_t* user_data

char ***header_key**

For HTTP_EVENT_ON_HEADER event_id, it's store current http header key

char ***header_value**

For HTTP_EVENT_ON_HEADER event_id, it's store current http header value

struct **esp_http_client_config_t**

HTTP configuration.

Public Members

const char ***url**

HTTP URL, the information on the URL is most important, it overrides the other fields below, if any

const char ***host**

Domain or IP as string

int **port**

Port to connect, default depend on esp_http_client_transport_t (80 or 443)

const char ***username**

Using for Http authentication

const char ***password**

Using for Http authentication

esp_http_client_auth_type_t **auth_type**

Http authentication type, see `esp_http_client_auth_type_t`

const char ***path**

HTTP Path, if not set, default is `/`

const char ***query**

HTTP query

const char ***cert_pem**

SSL server certification, PEM format as string, if the client requires to verify server

size_t **cert_len**

Length of the buffer pointed to by cert_pem. May be 0 for null-terminated pem

const char ***client_cert_pem**

SSL client certification, PEM format as string, if the server requires to verify client

size_t **client_cert_len**

Length of the buffer pointed to by client_cert_pem. May be 0 for null-terminated pem

const char ***client_key_pem**

SSL client key, PEM format as string, if the server requires to verify client

size_t **client_key_len**

Length of the buffer pointed to by client_key_pem. May be 0 for null-terminated pem

const char ***client_key_password**

Client key decryption password string

size_t **client_key_password_len**

String length of the password pointed to by client_key_password

const char ***user_agent**

The User Agent string to send with HTTP requests

esp_http_client_method_t **method**

HTTP Method

int **timeout_ms**

Network timeout in milliseconds

bool **disable_auto_redirect**

Disable HTTP automatic redirects

int **max_redirection_count**

Max number of redirections on receiving HTTP redirect status code, using default value if zero

int **max_authorization_retries**

Max connection retries on receiving HTTP unauthorized status code, using default value if zero. Disables authorization retry if -1

http_event_handle_cb **event_handler**

HTTP Event Handle

esp_http_client_transport_t **transport_type**

HTTP transport type, see `esp_http_client_transport_t`

int **buffer_size**

HTTP receive buffer size

int **buffer_size_tx**

HTTP transmit buffer size

void ***user_data**

HTTP user_data context

bool **is_async**

Set asynchronous mode, only supported with HTTPS for now

bool **use_global_ca_store**

Use a global ca_store for all the connections in which this bool is set.

bool **skip_cert_common_name_check**

Skip any validation of server certificate CN field

esp_err_t (*__crt_bundle_attach__)(void *conf)

Function pointer to esp_crt_bundle_attach. Enables the use of certification bundle for server verification, must be enabled in menuconfig

bool __keep_alive_enable__

Enable keep-alive timeout

int __keep_alive_idle__

Keep-alive idle time. Default is 5 (second)

int __keep_alive_interval__

Keep-alive interval time. Default is 5 (second)

int __keep_alive_count__

Keep-alive packet retry send count. Default is 3 counts

struct ifreq *__if_name__

The name of interface for data to go through. Use the default interface without setting

Macros

__DEFAULT_HTTP_BUF_SIZE__

__ESP_ERR_HTTP_BASE__

Starting number of HTTP error codes

__ESP_ERR_HTTP_MAX_REDIRECT__

The error exceeds the number of HTTP redirects

__ESP_ERR_HTTP_CONNECT__

Error open the HTTP connection

__ESP_ERR_HTTP_WRITE_DATA__

Error write HTTP data

__ESP_ERR_HTTP_FETCH_HEADER__

Error read HTTP header from server

__ESP_ERR_HTTP_INVALID_TRANSPORT__

There are no transport support for the input scheme

__ESP_ERR_HTTP_CONNECTING__

HTTP connection hasn’t been established yet

__ESP_ERR_HTTP_EAGAIN__

Mapping of errno EAGAIN to esp_err_t

__ESP_ERR_HTTP_CONNECTION_CLOSED__

Read FIN from peer and the connection closed

Type Definitions

typedef struct esp_http_client ***esp_http_client_handle_t**

typedef struct *esp_http_client_event* ***esp_http_client_event_handle_t**

typedef struct *esp_http_client_event* **esp_http_client_event_t**

HTTP Client events data.

typedef *esp_err_t* (***http_event_handle_cb**)(*esp_http_client_event_t* *evt)

Enumerations

enum **esp_http_client_event_id_t**

HTTP Client events id.

Values:

enumerator **HTTP_EVENT_ERROR**

This event occurs when there are any errors during execution

enumerator **HTTP_EVENT_ON_CONNECTED**

Once the HTTP has been connected to the server, no data exchange has been performed

enumerator **HTTP_EVENT_HEADERS_SENT**

After sending all the headers to the server

enumerator **HTTP_EVENT_HEADER_SENT**

This header has been kept for backward compatability and will be deprecated in future versions esp-idf

enumerator **HTTP_EVENT_ON_HEADER**

Occurs when receiving each header sent from the server

enumerator **HTTP_EVENT_ON_DATA**

Occurs when receiving data from the server, possibly multiple portions of the packet

enumerator **HTTP_EVENT_ON_FINISH**

Occurs when finish a HTTP session

enumerator **HTTP_EVENT_DISCONNECTED**

The connection has been disconnected

enumerator **HTTP_EVENT_REDIRECT**

Intercepting HTTP redirects to handle them manually

enum **esp_http_client_transport_t**

HTTP Client transport.

Values:

enumerator **HTTP_TRANSPORT_UNKNOWN**

Unknown

enumerator **HTTP_TRANSPORT_OVER_TCP**

Transport over tcp

enumerator **HTTP_TRANSPORT_OVER_SSL**

Transport over ssl

enum **esp_http_client_method_t**

HTTP method.

Values:

enumerator **HTTP_METHOD_GET**

HTTP GET Method

enumerator **HTTP_METHOD_POST**

HTTP POST Method

enumerator **HTTP_METHOD_PUT**

HTTP PUT Method

enumerator **HTTP_METHOD_PATCH**

HTTP PATCH Method

enumerator **HTTP_METHOD_DELETE**

HTTP DELETE Method

enumerator **HTTP_METHOD_HEAD**

HTTP HEAD Method

enumerator **HTTP_METHOD_NOTIFY**

HTTP NOTIFY Method

enumerator **HTTP_METHOD_SUBSCRIBE**

HTTP SUBSCRIBE Method

enumerator **HTTP_METHOD_UNSUBSCRIBE**

HTTP UNSUBSCRIBE Method

enumerator **HTTP_METHOD_OPTIONS**

HTTP OPTIONS Method

enumerator **HTTP_METHOD_COPY**

HTTP COPY Method

enumerator **HTTP_METHOD_MOVE**

HTTP MOVE Method

enumerator **HTTP_METHOD_LOCK**

HTTP LOCK Method

enumerator **HTTP_METHOD_UNLOCK**

HTTP UNLOCK Method

enumerator **HTTP_METHOD_PROPFIND**

HTTP PROPFIND Method

enumerator **HTTP_METHOD_PROPPATCH**

HTTP PROPPATCH Method

enumerator **HTTP_METHOD_MKCOL**

HTTP MKCOL Method

enumerator **HTTP_METHOD_MAX**

enum **esp_http_client_auth_type_t**

HTTP Authentication type.

Values:

enumerator **HTTP_AUTH_TYPE_NONE**

No authention

enumerator **HTTP_AUTH_TYPE_BASIC**

HTTP Basic authentication

enumerator **HTTP_AUTH_TYPE_DIGEST**

HTTP Disgest authentication

enum **HttpStatus_Code**

Enum for the HTTP status codes.

Values:

enumerator **HttpStatus_Ok**

enumerator **HttpStatus_MultipleChoices**

enumerator **HttpStatus_MovedPermanently**

enumerator **HttpStatus_Found**

enumerator **HttpStatus_SeeOther**

enumerator **HttpStatus_TemporaryRedirect**

enumerator **HttpStatus_PermanentRedirect**

enumerator **HttpStatus_BadRequest**

enumerator **HttpStatus_Unauthorized**

enumerator **HttpStatus_Forbidden**

enumerator **HttpStatus_NotFound**

enumerator **HttpStatus_InternalError**

2.2.6 ESP Local Control

Overview

ESP Local Control (**esp_local_ctrl**) component in ESP-IDF provides capability to control an ESP device over Wi-Fi + HTTPS or BLE. It provides access to application defined **properties** that are available for reading / writing via a set of configurable handlers.

Initialization of the **esp_local_ctrl** service over BLE transport is performed as follows:

```
esp_local_ctrl_config_t config = {
    .transport = ESP_LOCAL_CTRL_TRANSPORT_BLE,
    .transport_config = {
        .ble = & (protocomm_ble_config_t) {
            .device_name  = SERVICE_NAME,
            .service_uuid = {
                /* LSB <---------------------------------------
                * ---------------------------------------> MSB */
                0x21, 0xd5, 0x3b, 0x8d, 0xbd, 0x75, 0x68, 0x8a,
                0xb4, 0x42, 0xeb, 0x31, 0x4a, 0x1e, 0x98, 0x3d
            }
        }
    },
    .proto_sec = {
        .version = PROTOCOM_SEC0,
        .custom_handle = NULL,
        .pop = NULL,
    },
    .handlers = {
        /* User defined handler functions */
        .get_prop_values = get_property_values,
        .set_prop_values = set_property_values,
        .usr_ctx         = NULL,
        .usr_ctx_free_fn = NULL
    },
    /* Maximum number of properties that may be set */
    .max_properties = 10
};

/* Start esp_local_ctrl service */
ESP_ERROR_CHECK(esp_local_ctrl_start(&config));
```

Similarly for HTTPS transport:

```
/* Set the configuration */
httpd_ssl_config_t https_conf = HTTPD_SSL_CONFIG_DEFAULT();

/* Load server certificate */
extern const unsigned char servercert_start[] asm("_binary_servercert_pem_
↪start");
```

(continues on next page)

(continued from previous page)

```
extern const unsigned char servercert_end[]   asm("_binary_servercert_pem_
↪end");
https_conf.servercert = servercert_start;
https_conf.servercert_len = servercert_end - servercert_start;

/* Load server private key */
extern const unsigned char prvtkey_pem_start[] asm("_binary_prvtkey_pem_
↪start");
extern const unsigned char prvtkey_pem_end[]   asm("_binary_prvtkey_pem_
↪end");
https_conf.prvtkey_pem = prvtkey_pem_start;
https_conf.prvtkey_len = prvtkey_pem_end - prvtkey_pem_start;

esp_local_ctrl_config_t config = {
    .transport = ESP_LOCAL_CTRL_TRANSPORT_HTTPD,
    .transport_config = {
        .httpd = &https_conf
    },
    .proto_sec = {
        .version = PROTOCOM_SEC0,
        .custom_handle = NULL,
        .pop = NULL,
    },
    .handlers = {
        /* User defined handler functions */
        .get_prop_values = get_property_values,
        .set_prop_values = set_property_values,
        .usr_ctx         = NULL,
        .usr_ctx_free_fn = NULL
    },
    /* Maximum number of properties that may be set */
    .max_properties = 10
};

/* Start esp_local_ctrl service */
ESP_ERROR_CHECK(esp_local_ctrl_start(&config));
```

You may set security for transport in ESP local control using following options:

1. *PROTOCOM_SEC2*: specifies that SRP6a based key exchange and end to end encryption based on AES-GCM is used. This is the most preffered option as it adds a robust security with Augmented PAKE protocol i.e. SRP6a.
2. *PROTOCOM_SEC1*: specifies that Curve25519 based key exchange and end to end encryption based on AES-CTR is used.
3. *PROTOCOM_SEC0*: specifies that data will be exchanged as a plain text (no security).
4. *PROTOCOM_SEC_CUSTOM*: you can define your own security requirement. Please note that you will also have to provide *custom_handle* of type *protocomm_security_t* * in this context.

Note: The respective security schemes need to be enabled through the project configuration menu. Please refer to the Enabling protocom security version section in *Protocol Communication* for more details.

Creating a property

Now that we know how to start the **esp_local_ctrl** service, let's add a property to it. Each property must have a unique *name* (string), a *type* (e.g. enum), *flags* (bit fields) and *size*.

The *size* is to be kept 0, if we want our property value to be of variable length (e.g. if its a string or bytestream). For fixed length property value data-types, like int, float, etc., setting the *size* field to the right value, helps **esp_local_ctrl** to perform internal checks on arguments received with write requests.

The interpretation of *type* and *flags* fields is totally upto the application, hence they may be used as enumerations, bit-fields, or even simple integers. One way is to use *type* values to classify properties, while *flags* to specify characteristics of a property.

Here is an example property which is to function as a timestamp. It is assumed that the application defines *TYPE_TIMESTAMP* and *READONLY*, which are used for setting the *type* and *flags* fields here.

```
/* Create a timestamp property */
esp_local_ctrl_prop_t timestamp = {
    .name         = "timestamp",
    .type         = TYPE_TIMESTAMP,
    .size         = sizeof(int32_t),
    .flags        = READONLY,
    .ctx          = func_get_time,
    .ctx_free_fn = NULL
};

/* Now register the property */
esp_local_ctrl_add_property(&timestamp);
```

Also notice that there is a ctx field, which is set to point to some custom *func_get_time()*. This can be used inside the property get / set handlers to retrieve timestamp.

Here is an example of *get_prop_values()* handler, which is used for retrieving the timestamp.

```
static esp_err_t get_property_values(size_t props_count,
                                     const esp_local_ctrl_prop_t *props,
                                     esp_local_ctrl_prop_val_t *prop_
↪values,
                                     void *usr_ctx)
{
    for (uint32_t i = 0; i < props_count; i++) {
        ESP_LOGI(TAG, "Reading %s", props[i].name);
        if (props[i].type == TYPE_TIMESTAMP) {
            /* Obtain the timer function from ctx */
            int32_t (*func_get_time)(void) = props[i].ctx;

            /* Use static variable for saving the value.
             * This is essential because the value has to be
             * valid even after this function returns.
             * Alternative is to use dynamic allocation
             * and set the free_fn field */
            static int32_t ts = func_get_time();
            prop_values[i].data = &ts;
        }
    }
    return ESP_OK;
}
```

Here is an example of *set_prop_values()* handler. Notice how we restrict from writing to read-only properties.

```
static esp_err_t set_property_values(size_t props_count,
                                     const esp_local_ctrl_prop_t *props,
                                     const esp_local_ctrl_prop_val_t␣
↪*prop_values,
                                     void *usr_ctx)
{
    for (uint32_t i = 0; i < props_count; i++) {
        if (props[i].flags & READONLY) {
            ESP_LOGE(TAG, "Cannot write to read-only property %s",␣
↪props[i].name);
            return ESP_ERR_INVALID_ARG;
        } else {
```

(continues on next page)

(continued from previous page)

```
            ESP_LOGI(TAG, "Setting %s", props[i].name);

            /* For keeping it simple, lets only log the incoming data */
            ESP_LOG_BUFFER_HEX_LEVEL(TAG, prop_values[i].data,
                                     prop_values[i].size, ESP_LOG_INFO);
        }
    }
    return ESP_OK;
}
```

For complete example see protocols/esp_local_ctrl

Client Side Implementation

The client side implementation will have establish a protocomm session with the device first, over the supported mode of transport, and then send and receive protobuf messages understood by the **esp_local_ctrl** service. The service will translate these messages into requests and then call the appropriate handlers (set / get). Then, the generated response for each handler is again packed into a protobuf message and transmitted back to the client.

See below the various protobuf messages understood by the **esp_local_ctrl** service:

1. *get_prop_count* : This should simply return the total number of properties supported by the service
2. *get_prop_values* : This accepts an array of indices and should return the information (name, type, flags) and values of the properties corresponding to those indices
3. *set_prop_values* : This accepts an array of indices and an array of new values, which are used for setting the values of the properties corresponding to the indices

Note that indices may or may not be the same for a property, across multiple sessions. Therefore, the client must only use the names of the properties to uniquely identify them. So, every time a new session is established, the client should first call *get_prop_count* and then *get_prop_values*, hence form an index to name mapping for all properties. Now when calling *set_prop_values* for a set of properties, it must first convert the names to indexes, using the created mapping. As emphasized earlier, the client must refresh the index to name mapping every time a new session is established with the same device.

The various protocomm endpoints provided by **esp_local_ctrl** are listed below:

Table 1: Endpoints provided by ESP Local Control

Endpoint Name (BLE + GATT Server)	URI (HTTPS Server + mDNS)	Description
esp_local_ctrl/version	https://<mdns-hostname>.local/esp_local_ctrl/version	Endpoint used for retrieving version string
esp_local_ctrl/control	https://<mdns-hostname>.local/esp_local_ctrl/control	Endpoint used for sending / receiving control messages

API Reference

Header File

- components/esp_local_ctrl/include/esp_local_ctrl.h

Functions

const *esp_local_ctrl_transport_t* ***esp_local_ctrl_get_transport_ble**(void)

Function for obtaining BLE transport mode.

const *esp_local_ctrl_transport_t* ***esp_local_ctrl_get_transport_httpd**(void)

Function for obtaining HTTPD transport mode.

esp_err_t **esp_local_ctrl_start**(const *esp_local_ctrl_config_t* *config)

Start local control service.

Parameters **config** –**[in]** Pointer to configuration structure

Returns

- ESP_OK : Success
- ESP_FAIL : Failure

esp_err_t **esp_local_ctrl_stop**(void)

Stop local control service.

esp_err_t **esp_local_ctrl_add_property**(const *esp_local_ctrl_prop_t* *prop)

Add a new property.

This adds a new property and allocates internal resources for it. The total number of properties that could be added is limited by configuration option `max_properties`

Parameters **prop** –**[in]** Property description structure

Returns

- ESP_OK : Success
- ESP_FAIL : Failure

esp_err_t **esp_local_ctrl_remove_property**(const char *name)

Remove a property.

This finds a property by name, and releases the internal resources which are associated with it.

Parameters **name** –**[in]** Name of the property to remove

Returns

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Failure

const *esp_local_ctrl_prop_t* ***esp_local_ctrl_get_property**(const char *name)

Get property description structure by name.

This API may be used to get a property' s context structure `esp_local_ctrl_prop_t` when its name is known

Parameters **name** –**[in]** Name of the property to find

Returns

- Pointer to property
- NULL if not found

esp_err_t **esp_local_ctrl_set_handler**(const char *ep_name, *protocomm_req_handler_t* handler, void *user_ctx)

Register protocomm handler for a custom endpoint.

This API can be called by the application to register a protocomm handler for an endpoint after the local control service has started.

Note: In case of BLE transport the names and uuids of all custom endpoints must be provided beforehand as a part of the `protocomm_ble_config_t` structure set in `esp_local_ctrl_config_t`, and passed to `esp_local_ctrl_start()`.

Parameters

- **ep_name** –**[in]** Name of the endpoint
- **handler** –**[in]** Endpoint handler function
- **user_ctx** –**[in]** User data

Returns

- ESP_OK : Success
- ESP_FAIL : Failure

Unions

union **esp_local_ctrl_transport_config_t**

#include <esp_local_ctrl.h> Transport mode (BLE / HTTPD) configuration.

Public Members

esp_local_ctrl_transport_config_ble_t ***ble**

This is same as `protocomm_ble_config_t`. See `protocomm_ble.h` for available configuration parameters.

esp_local_ctrl_transport_config_httpd_t ***httpd**

This is same as `httpd_ssl_config_t`. See `esp_https_server.h` for available configuration parameters.

Structures

struct **esp_local_ctrl_prop**

Property description data structure, which is to be populated and passed to the `esp_local_ctrl_add_property()` function.

Once a property is added, its structure is available for read-only access inside `get_prop_values()` and `set_prop_values()` handlers.

Public Members

char ***name**

Unique name of property

uint32_t **type**

Type of property. This may be set to application defined enums

size_t **size**

Size of the property value, which:

- if zero, the property can have values of variable size
- if non-zero, the property can have values of fixed size only, therefore, checks are performed internally by esp_local_ctrl when setting the value of such a property

uint32_t **flags**

Flags set for this property. This could be a bit field. A flag may indicate property behavior, e.g. read-only / constant

void ***ctx**

Pointer to some context data relevant for this property. This will be available for use inside the `get_prop_values` and `set_prop_values` handlers as a part of this property structure. When set, this is valid throughout the lifetime of a property, till either the property is removed or the esp_local_ctrl service is stopped.

void (*`ctx_free_fn`)(void *ctx)

Function used by esp_local_ctrl to internally free the property context when `esp_local_ctrl_remove_property()` or `esp_local_ctrl_stop()` is called.

struct `esp_local_ctrl_prop_val`

Property value data structure. This gets passed to the `get_prop_values()` and `set_prop_values()` handlers for the purpose of retrieving or setting the present value of a property.

Public Members

void *`data`

Pointer to memory holding property value

size_t `size`

Size of property value

void (*`free_fn`)(void *data)

This may be set by the application in `get_prop_values()` handler to tell `esp_local_ctrl` to call this function on the data pointer above, for freeing its resources after sending the `get_prop_values` response.

struct `esp_local_ctrl_handlers`

Handlers for receiving and responding to local control commands for getting and setting properties.

Public Members

esp_err_t (*`get_prop_values`)(size_t props_count, const *esp_local_ctrl_prop_t* props[], *esp_local_ctrl_prop_val_t* prop_values[], void *usr_ctx)

Handler function to be implemented for retrieving current values of properties.

Note: If any of the properties have fixed sizes, the size field of corresponding element in `prop_values` need to be set

---

Param props_count **[in]** Total elements in the props array

Param props **[in]** Array of properties, the current values for which have been requested by the client

Param prop_values **[out]** Array of empty property values, the elements of which need to be populated with the current values of those properties specified by props argument

Param usr_ctx **[in]** This provides value of the `usr_ctx` field of `esp_local_ctrl_handlers_t` structure

Return Returning different error codes will convey the corresponding protocol level errors to the client :

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : InvalidArgument
- ESP_ERR_INVALID_STATE : InvalidProto
- All other error codes : InternalError

esp_err_t (*`set_prop_values`)(size_t props_count, const *esp_local_ctrl_prop_t* props[], const *esp_local_ctrl_prop_val_t* prop_values[], void *usr_ctx)

Handler function to be implemented for changing values of properties.

Note: If any of the properties have variable sizes, the size field of the corresponding element in `prop_values` must be checked explicitly before making any assumptions on the size.

Param props_count [in] Total elements in the props array
Param props [in] Array of properties, the values for which the client requests to change
Param prop_values [in] Array of property values, the elements of which need to be used for updating those properties specified by props argument
Param usr_ctx [in] This provides value of the `usr_ctx` field of `esp_local_ctrl_handlers_t` structure
Return Returning different error codes will convey the corresponding protocol level errors to the client :
- ESP_OK : Success
- ESP_ERR_INVALID_ARG : InvalidArgument
- ESP_ERR_INVALID_STATE : InvalidProto
- All other error codes : InternalError

void ***usr_ctx**

Context pointer to be passed to above handler functions upon invocation. This is different from the property level context, as this is valid throughout the lifetime of the `esp_local_ctrl` service, and freed only when the service is stopped.

void (***usr_ctx_free_fn**)(void *usr_ctx)

Pointer to function which will be internally invoked on `usr_ctx` for freeing the context resources when `esp_local_ctrl_stop()` is called.

struct **esp_local_ctrl_proto_sec_cfg**

Protocom security configs

Public Members

esp_local_ctrl_proto_sec_t **version**

This sets protocom security version, sec0/sec1 or custom If custom, user must provide handle via `proto_sec_custom_handle` below

void ***custom_handle**

Custom security handle if security is set custom via `proto_sec` above This handle must follow `protocomm_security_t` signature

const void ***pop**

Proof of possession to be used for local control. Could be NULL.

const void ***sec_params**

Pointer to security params (NULL if not needed). This is not needed for protocomm security 0 This pointer should hold the struct of type esp_local_ctrl_security1_params_t for protocomm security 1 and esp_local_ctrl_security2_params_t for protocomm security 2 respectively. Could be NULL.

struct **esp_local_ctrl_config**

Configuration structure to pass to `esp_local_ctrl_start()`

Public Members

const *esp_local_ctrl_transport_t* ***transport**

Transport layer over which service will be provided

esp_local_ctrl_transport_config_t **transport_config**

Transport layer over which service will be provided

esp_local_ctrl_proto_sec_cfg_t **proto_sec**

Security version and POP

esp_local_ctrl_handlers_t **handlers**

Register handlers for responding to get/set requests on properties

size_t **max_properties**

This limits the number of properties that are available at a time

Macros

ESP_LOCAL_CTRL_TRANSPORT_BLE

ESP_LOCAL_CTRL_TRANSPORT_HTTPD

Type Definitions

typedef struct *esp_local_ctrl_prop* **esp_local_ctrl_prop_t**

Property description data structure, which is to be populated and passed to the `esp_local_ctrl_add_property()` function.

Once a property is added, its structure is available for read-only access inside `get_prop_values()` and `set_prop_values()` handlers.

typedef struct *esp_local_ctrl_prop_val* **esp_local_ctrl_prop_val_t**

Property value data structure. This gets passed to the `get_prop_values()` and `set_prop_values()` handlers for the purpose of retrieving or setting the present value of a property.

typedef struct *esp_local_ctrl_handlers* **esp_local_ctrl_handlers_t**

Handlers for receiving and responding to local control commands for getting and setting properties.

typedef struct esp_local_ctrl_transport **esp_local_ctrl_transport_t**

Transport mode (BLE / HTTPD) over which the service will be provided.

This is forward declaration of a private structure, implemented internally by `esp_local_ctrl`.

typedef struct *protocomm_ble_config* **esp_local_ctrl_transport_config_ble_t**

Configuration for transport mode BLE.

This is a forward declaration for `protocomm_ble_config_t`. To use this, application must set CONFIG_BT_BLUEDROID_ENABLED and include `protocomm_ble.h`.

typedef struct *httpd_config* **esp_local_ctrl_transport_config_httpd_t**

Configuration for transport mode HTTPD.

This is a forward declaration for `httpd_ssl_config_t` (for HTTPS) or `httpd_config_t` (for HTTP)

typedef enum *esp_local_ctrl_proto_sec* **esp_local_ctrl_proto_sec_t**

Security types for esp_local_control.

typedef *protocomm_security1_params_t* **esp_local_ctrl_security1_params_t**

typedef *protocomm_security2_params_t* **esp_local_ctrl_security2_params_t**

typedef struct *esp_local_ctrl_proto_sec_cfg* **esp_local_ctrl_proto_sec_cfg_t**

Protocom security configs

typedef struct *esp_local_ctrl_config* **esp_local_ctrl_config_t**

Configuration structure to pass to `esp_local_ctrl_start()`

Enumerations

enum **esp_local_ctrl_proto_sec**

Security types for esp_local_control.

Values:

enumerator **PROTOCOM_SEC0**

enumerator **PROTOCOM_SEC1**

enumerator **PROTOCOM_SEC2**

enumerator **PROTOCOM_SEC_CUSTOM**

2.2.7 ESP Serial Slave Link

Overview

Espressif provides several chips that can work as slaves. These slave devices rely on some common buses, and have their own communication protocols over those buses. The *esp_serial_slave_link* component is designed for the master to communicate with ESP slave devices through those protocols over the bus drivers.

After an *esp_serial_slave_link* device is initialized properly, the application can use it to communicate with the ESP slave devices conveniently.

Espressif Device protocols

For more details about Espressif device protocols, see the following documents.

Communication with ESP SDIO Slave This document describes the process of initialization of an ESP SDIO Slave device and then provides details on the ESP SDIO Slave protocol - a non-standard protocol that allows an SDIO Host to communicate with an ESP SDIO slave.

The ESP SDIO Slave protocol was created to implement the communication between SDIO host and slave, because the SDIO specification only shows how to access the custom region of a card (by sending CMD52 and CMD53 to Functions 1-7) without any details regarding the underlying hardware implementation.

SDIO Slave Capabilities of Espressif chips The services provided by the SDIO Slave peripheral of the ESP32 chip are listed in the table below:

Services	ESP32
SDIO slave	Y
Tohost intr	8
Frhost intr	8
TX DMA	Y
RX DMA	Y
Shared registers	56*

- * Not including the interrupt registers

ESP SDIO Slave Initialization The host should initialize the ESP32 SDIO slave according to the standard SDIO initialization process (Section 3.1.2 of SDIO Simplified Specification). In this specification as well as below, the SDIO slave is called an (SD)IO card. Here is a brief example of an ESP SDIO Slave initialization process:

1. **SDIO reset** CMD52 (Write 0x6=0x8)
2. **SD reset** CMD0
3. **Check whether IO card (optional)** CMD8
4. **Send SDIO op cond and wait for card ready** CMD5 arg = 0x00000000
   CMD5 arg = 0x00ff8000 (according to the response above, poll until ready)
   Example: Arg of R4 after first CMD5 (arg=0x00000000) is 0xXXFFFF00.
   Keep sending CMD5 with arg=0x00FFFF00 until the R4 shows card ready (arg bit 31=1).
5. **Set address** CMD3
6. **Select card** CMD7 (arg address according to CMD3 response)
   Example: Arg of R6 after CMD3 is 0x0001xxxx.
   Arg of CMD7 should be 0x00010000.
7. **Select 4-bit mode (optional)** CMD52 (Write 0x07=0x02)
8. **Enable func1** CMD52 (Write 0x02=0x02)
9. **Enable SDIO interrupt (required if interrupt line (DAT1) is used)** CMD52 (Write 0x04=0x03)
10. **Set Func0 blocksize (optional, default value is 512 (0x200))** CMD52/53 (Read 0x10~0x11)
    CMD52/53 (Write 0x10=0x00)
    CMD52/53 (Write 0x11=0x02)
    CMD52/53 (Read 0x10~0x11, read to check the final value)
11. **Set Func1 blocksize (optional, default value is 512 (0x200))** CMD52/53 (Read 0x110~0x111)
    CMD52/53 (Write 0x110=0x00)
    CMD52/53 (Write 0x111=0x02)
    CMD52/53 (Read 0x110~0x111, read to check the final value)

ESP SDIO Slave Protocol The ESP SDIO Slave protocol is based on the SDIO Specification's I/O Read/Write commands, i.e., CMD52 and CMD53. The protocol offers the following services:

- Sending FIFO and receiving FIFO
- 52 8-bit R/W registers shared by host and slave (For details, see *ESP32 Technical Reference Manual* > *SDIO Slave Controller* > *Register Summary* > SDIO SLC Host registers [PDF]
- 16 general purpose interrupt sources, 8 from host to slave and 8 from slave to host

To begin communication, the host needs to enable the I/O Function 1 in the slave and access its registers as described below.

Check the code example peripherals/sdio.

The *ESP Serial Slave Link* component implements the logic of this protocol for ESP32 SDIO Host when communicating with an ESP32 SDIO slave.

Slave register table

32-bit

- 0x044 (TOKEN_RDATA): in which bit 27-16 holds the number of the receiving buffer.
- 0x058 (INT_ST): holds the interrupt source bits from slave to host.
- 0x060 (PKT_LEN): holds the accumulated data length (in bytes) already read by host plus the data copied to the buffer but yet to be read.
- 0x0D4 (INT_CLR): write 1 to clear interrupt bits corresponding to INT_ST.
- 0x0DC (INT_ENA): mask bits for interrupts from slave to host.

8-bit Shared general purpose registers:

- 0x06C-0x077: R/W registers 0-11 shared by slave and host.
- 0x07A-0x07B: R/W registers 14-15 shared by slave and host.
- 0x07E-0x07F: R/W registers 18-19 shared by slave and host.
- 0x088-0x08B: R/W registers 24-27 shared by slave and host.
- 0x09C-0x0BB: R/W registers 32-63 shared by slave and host.

Interrupt Registers: - 0x08D (SLAVE_INT): bits for host to interrupt slave. auto clear.

FIFO (sending and receiving) 0x090 - 0x1F7FF are reserved for FIFOs.

The address of CMD53 is related to the length requested to read from or write to the slave in a single transfer, as demonstrated by the equation below:

requested length = 0x1F800-address

The slave will respond with data that has a length equal to the length field of CMD53. In cases where the data is longer than the *requested length*, the data will be zero filled (when sending) or discarded (when receiving). This includes both the block and the byte mode of CMD53.

Note: The function number should be set to 1, OP Code should be set to 1 (for CMD53).

In order to achieve higher efficiency when accessing the FIFO by an arbitrary length, the block and byte modes of CMD53 can be used in combination. For example, given that the block size is set to 512 by default, you can write/get 1031 bytes of data from the FIFO by doing the following:

1. Send CMD53 in block mode, block count=2 (1024 bytes) to address 0x1F3F9=0x1F800-**1031**.
2. Then send CMD53 in byte mode, byte count=8 (or 7 if your controller supports that) to address 0x1F7F9=0x1F800-**7**.

Interrupts SDIO interrupts are "level sensitive". For host interrupts, the slave sends an interrupt by pulling the DAT1 line down at a proper time. The host detects when the interrupt line is pulled down and reads the INT_ST register to determine the source of the interrupt. After that, the host can clear the interrupt bits by writing the INT_CLR register and process the interrupt. The host can also mask unneeded sources by clearing the bits in the INT_ENA register corresponding to the sources. If all the sources are cleared (or masked), the DAT1 line goes inactive.

On ESP32, the corresponding host_int bits are: bit 0 to bit 7.

For slave interrupts, the host sends a transfer to write the SLAVE_INT register. Once a bit is set to 1, the slave hardware and the driver will detect it and inform the application.

Receiving FIFO To write to the slave's receiving FIFO, the host should complete the following steps:

1. **Read the TOKEN1 field (bits 27-16) of the register TOKEN_RDATA (0x044)**. The buffer number remaining is TOKEN1 minus the number of buffers used by host.
2. **Make sure the buffer number is sufficient** (*buffer_size* x *buffer_num* is greater than the data to write, *buffer_size* is pre-defined between the host and the slave before the communication starts). Otherwise, keep returning to Step 1 until the buffer size is sufficient.

3. **Write to the FIFO address with CMD53**. Note that the *requested length* should not exceed the length calculated at Step 2, and the FIFO address is related to *requested length*.
4. **Calculate used buffers**. Note that a partially used buffer at the tail is counted as used.

Sending FIFO To read the slave's sending FIFO, the host should complete the following steps:

1. **Wait for the interrupt line to become active** (optional, low by default).
2. **Read (poll) the interrupt bits in the INT_ST register** to monitor if new packets exist.
3. **If new packets are ready, read the PKT_LEN register**. Before reading the packets, determine the length of data to be read. As the host keeps the length of data already read from the slave, subtract this value from PKT_LEN, the result will be the maximum length of data available for reading. If no data has been added to the sending FIFO yet, wait and poll until the slave is ready and update PKT_LEN.
4. **Read from the FIFO using CMD53**. Note that the *requested length* should not be greater than calculated at Step 3, and the FIFO address is related to *requested length*.
5. **Update the read length**.

Warning: The driver for ESP32 hasn't been developed yet.

ESP SPI Slave HD (Half Duplex) Mode Protocol

SPI Slave Capabilities of Espressif chips

	ESP32	ESP32-S2	ESP32-C3
SPI Slave HD	N	Y (v2)	Y (v2)
Tohost intr		N	N
Frhost intr		2 *	2 *
TX DMA		Y	Y
RX DMA		Y	Y
Shared registers		72	64

Introduction In the half duplex mode, the master has to use the protocol defined by the slave to communicate with the slave. Each transaction may consist of the following phases (list by the order they should exist):

- Command: 8-bit, master to slave
  This phase determines the rest phases of the transactions. See *Supported Commands*.
- Address: 8-bit, master to slave, optional
  For some commands (WRBUF, RDBUF), this phase specifies the address of the shared buffer to write to/read from. For other commands with this phase, they are meaningless but still have to exist in the transaction.
- Dummy: 8-bit, floating, optional
  This phase is the turnaround time between the master and the slave on the bus, and also provides enough time for the slave to prepare the data to send to the master.
- Data: variable length, the direction is also determined by the command.
  This may be a data OUT phase, in which the direction is slave to master, or a data IN phase, in which the direction is master to slave.

The *direction* means which side (master or slave) controls the MOSI, MISO, WP, and HD pins.

Data IO Modes In some IO modes, more data wires can be used to send the data. As a result, the SPI clock cycles required for the same amount of data will be less than in the 1-bit mode. For example, in QIO mode, address and data (IN and OUT) should be sent on all 4 data wires (MOSI, MISO, WP, and HD). Here are the modes supported by the ESP32-S2 SPI slave and the wire number used in corresponding modes.

Mode	command WN	address WN	dummy cycles	data WN
1-bit	1	1	1	1
DOUT	1	1	4	2
DIO	1	2	4	2
QOUT	1	1	4	4
QIO	1	4	4	4
QPI	4	4	4	4

Normally, which mode is used is determined by the command sent by the master (See *Supported Commands*), except the QPI mode.

QPI Mode The QPI mode is a special state of the SPI Slave. The master can send the ENQPI command to put the slave into the QPI mode state. In the QPI mode, the command is also sent in 4-bit, thus it ' s not compatible with the normal modes. The master should only send QPI commands when the slave is in QPI mode. To exit from the QPI mode, master can send the EXQPI command.

Supported Commands

Note: The command name is in a master-oriented direction. For example, WRBUF means master writes the buffer of slave.

Name	Description	Command	Address	Data
WRBUF	Write buffer	0x01	Buf addr	master to slave, no longer than buffer size
RDBUF	Read buffer	0x02	Buf addr	slave to master, no longer than buffer size
WRDMA	Write DMA	0x03	8 bits	master to slave, no longer than length provided by slave
RDDMA	Read DMA	0x04	8 bits	slave to master, no longer than length provided by slave
SEG_DONE	Segments done	0x05	•	•
ENQPI	Enter QPI mode	0x06	•	•
WR_DONE	Write segments done	0x07	•	•
CMD8	Interrupt	0x08	•	•
CMD9	Interrupt	0x09	•	•
CMDA	Interrupt	0x0A	•	•
EXQPI	Exit QPI mode	0xDD	•	•

Moreover, WRBUF, RDBUF, WRDMA, RDDMA commands have their 2-bit and 4-bit version. To do transactions in 2-bit or 4-bit mode, send the original command ORed by the corresponding command mask below. For example, command 0xA1 means WRBUF in QIO mode.

Mode	Mask
1-bit	0x00
DOUT	0x10
DIO	0x50
QOUT	0x20
QIO	0xA0
QPI	0xA0

Segment Transaction Mode Segment transaction mode is the only mode supported by the SPI Slave HD driver for now. In this mode, for a transaction the slave load onto the DMA, the master is allowed to read or write in segments. This way the master doesn't have to prepare a large buffer as the size of data provided by the slave. After the master finishes reading/writing a buffer, it has to send the corresponding termination command to the slave as a synchronization signal. The slave driver will update new data (if exist) onto the DMA upon seeing the termination command.

The termination command is WR_DONE (0x07) for the WRDMA and CMD8 (0x08) for the RDDMA.

Here's an example for the flow the master read data from the slave DMA:

1. The slave loads 4092 bytes of data onto the RDDMA
2. The master do seven RDDMA transactions, each of them is 512 bytes long, and reads the first 3584 bytes from the slave
3. The master do the last RDDMA transaction of 512 bytes (equal, longer, or shorter than the total length loaded by the slave are all allowed). The first 508 bytes are valid data from the slave, while the last 4 bytes are meaningless bytes.
4. The master sends CMD8 to the slave
5. The slave loads another 4092 bytes of data onto the RDDMA
6. The master can start new reading transactions after it sends the CMD8

Terminology

- ESSL: Abbreviation for ESP Serial Slave Link, the component described by this document.
- Master: The device running the *esp_serial_slave_link* component.
- ESSL device: a virtual device on the master associated with an ESP slave device. The device context has the knowledge of the slave protocol above the bus, relying on some bus drivers to communicate with the slave.
- ESSL device handle: a handle to ESSL device context containing the configuration, status and data required by the ESSL component. The context stores the driver configurations, communication state, data shared by master and slave, etc.
  The context should be initialized before it is used, and get deinitialized if not used any more. The master application operates on the ESSL device through this handle.
- ESP slave: the slave device connected to the bus, which ESSL component is designed to communicate with.
- Bus: The bus over which the master and the slave communicate with each other.
- Slave protocol: The special communication protocol specified by Espressif HW/SW over the bus.
- TX buffer num: a counter, which is on the slave and can be read by the master, indicates the accumulated buffer numbers that the slave has loaded to the hardware to receive data from the master.
- RX data size: a counter, which is on the slave and can be read by the master, indicates the accumulated data size that the slave has loaded to the hardware to send to the master.

Services provided by ESP slave

There are some common services provided by the Espressif slaves:

1. Tohost Interrupts: The slave can inform the master about certain events by the interrupt line. (optional)
2. Frhost Interrupts: The master can inform the slave about certain events.
3. Tx FIFO (master to slave): the slave can send data in stream to the master. The SDIO slave can also indicate it has new data to send to master by the interrupt line.

The slave updates the TX buffer num to inform the master how much data it can receive, and the master then read the TX buffer num, and take off the used buffer number to know how many buffers are remaining.

4. Rx FIFO (slave to master): the slave can receive data from the master in units of receiving buffers.
   The slave updates the RX data size to inform the master how much data it has prepared to send, and then the master read the data size, and take off the data length it has already received to know how many data is remaining.
5. Shared registers: the master can read some part of the registers on the slave, and also write these registers to let the slave read.

The services provided by the slave depends on the slave' s model. See *SDIO Slave Capabilities of Espressif chips* and *SPI Slave Capabilities of Espressif chips* for more details.

Initialization of ESP Serial Slave Link

ESP SDIO Slave The ESP SDIO slave link (ESSL SDIO) devices relies on the sdmmc component. It includes the usage of communicating with ESP SDIO Slave device via SDSPI feature. The ESSL device should be initialized as below:

1. Initialize a sdmmc card (see :doc:` Document of SDMMC driver </api-reference/storage/sdmmc>`) structure.
2. Call `sdmmc_card_init()` to initialize the card.
3. Initialize the ESSL device with `essl_sdio_config_t`. The *card* member should be the `sdmmc_card_t` got in step 2, and the *recv_buffer_size* member should be filled correctly according to pre-negotiated value.
4. Call `essl_init()` to do initialization of the SDIO part.
5. Call `essl_wait_for_ready()` to wait for the slave to be ready.

ESP SPI Slave

Note: If you are communicating with the ESP SDIO Slave device through SPI interface, you should use the *SDIO interface* instead.

Hasn' t been supported yet.

APIs

After the initialization process above is performed, you can call the APIs below to make use of the services provided by the slave:

Tohost Interrupts (optional)

1. Call `essl_get_intr_ena()` to know which events will trigger the interrupts to the master.
2. Call `essl_set_intr_ena()` to set the events that will trigger interrupts to the master.
3. Call `essl_wait_int()` to wait until interrupt from the slave, or timeout.
4. When interrupt is triggered, call `essl_get_intr()` to know which events are active, and call `essl_clear_intr()` to clear them.

Frhost Interrupts

1. Call `essl_send_slave_intr()` to trigger general purpose interrupt of the slave.

TX FIFO

1. Call `essl_get_tx_buffer_num()` to know how many buffers the slave has prepared to receive data from the master. This is optional. The master will poll *tx_buffer_num* when it try to send packets to the slave, until the slave has enough buffer or timeout.
2. Call `essl_send_packet()` to send data to the slave.

RX FIFO

1. Call `essl_get_rx_data_size()` to know how many data the slave has prepared to send to the master. This is optional. When the master tries to receive data from the slave, it will update the *rx_data_size* for once, if the current *rx_data_size* is shorter than the buffer size the master prepared to receive. And it may poll the *rx_data_size* if the *rx_dat_size* keeps 0, until timeout.
2. Call `essl_get_packet()` to receive data from the slave.

Reset counters (Optional) Call `essl_reset_cnt()` to reset the internal counter if you find the slave has reset its counter.

Application Example

The example below shows how ESP32 SDIO host and slave communicate with each other. The host use the ESSL SDIO.

peripherals/sdio.

Please refer to the specific example README.md for details.

API Reference

Header File

- components/driver/test/esp_serial_slave_link/include/esp_serial_slave_link/essl.h

Functions

esp_err_t **essl_init**(*essl_handle_t* handle, uint32_t wait_ms)

Initialize the slave.

Parameters
- **handle** –Handle of an ESSL device.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: If success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- Other value returned from lower layer `init`.

esp_err_t **essl_wait_for_ready**(*essl_handle_t* handle, uint32_t wait_ms)

Wait for interrupt of an ESSL slave device.

Parameters
- **handle** –Handle of an ESSL device.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: If success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

esp_err_t **essl_get_tx_buffer_num**(*essl_handle_t* handle, uint32_t *out_tx_num, uint32_t wait_ms)

Get buffer num for the host to send data to the slave. The buffers are size of `buffer_size`.

Parameters
- **handle** –Handle of a ESSL device.
- **out_tx_num** –Output of buffer num that host can send data to ESSL slave.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode

- One of the error codes from SDMMC/SPI host controller

esp_err_t **essl_get_rx_data_size**(*essl_handle_t* handle, uint32_t *out_rx_size, uint32_t wait_ms)

Get the size, in bytes, of the data that the ESSL slave is ready to send

Parameters
- **handle** –Handle of an ESSL device.
- **out_rx_size** –Output of data size to read from slave, in bytes
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller

esp_err_t **essl_reset_cnt**(*essl_handle_t* handle)

Reset the counters of this component. Usually you don't need to do this unless you know the slave is reset.

Parameters **handle** –Handle of an ESSL device.

Returns
- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- ESP_ERR_INVALID_ARG: Invalid argument, handle is not init.

esp_err_t **essl_send_packet**(*essl_handle_t* handle, const void *start, size_t length, uint32_t wait_ms)

Send a packet to the ESSL Slave. The Slave receives the packet into buffers whose size is `buffer_size` (configured during initialization).

Parameters
- **handle** –Handle of an ESSL device.
- **start** –Start address of the packet to send
- **length** –Length of data to send, if the packet is over-size, the it will be divided into blocks and hold into different buffers automatically.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG: Invalid argument, handle is not init or other argument is not valid.
- ESP_ERR_TIMEOUT: No buffer to use, or error ftrom SDMMC host controller.
- ESP_ERR_NOT_FOUND: Slave is not ready for receiving.
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller.

esp_err_t **essl_get_packet**(*essl_handle_t* handle, void *out_data, size_t size, size_t *out_length, uint32_t wait_ms)

Get a packet from ESSL slave.

Parameters
- **handle** –Handle of an ESSL device.
- **out_data** –**[out]** Data output address
- **size** –The size of the output buffer, if the buffer is smaller than the size of data to receive from slave, the driver returns `ESP_ERR_NOT_FINISHED`
- **out_length** –**[out]** Output of length the data actually received from slave.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK Success: All the data has been read from the slave.
- ESP_ERR_INVALID_ARG: Invalid argument, The handle is not initialized or the other arguments are invalid.
- ESP_ERR_NOT_FINISHED: Read was successful, but there is still data remaining.
- ESP_ERR_NOT_FOUND: Slave is not ready to send data.
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller.

esp_err_t **essl_write_reg** (*essl_handle_t* handle, uint8_t addr, uint8_t value, uint8_t *value_o, uint32_t wait_ms)

Write general purpose R/W registers (8-bit) of ESSL slave.

Note: sdio 28-31 are reserved, the lower API helps to skip.

Parameters
- **handle** –Handle of an ESSL device.
- **addr** –Address of register to write. For SDIO, valid address: 0-59. For SPI, see `essl_spi.h`
- **value** –Value to write to the register.
- **value_o** –Output of the returned written value.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK Success
- One of the error codes from SDMMC/SPI host controller

esp_err_t **essl_read_reg** (*essl_handle_t* handle, uint8_t add, uint8_t *value_o, uint32_t wait_ms)

Read general purpose R/W registers (8-bit) of ESSL slave.

Parameters
- **handle** –Handle of a `essl` device.
- **add** –Address of register to read. For SDIO, Valid address: 0-27, 32-63 (28-31 reserved, return interrupt bits on read). For SPI, see `essl_spi.h`
- **value_o** –Output value read from the register.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK Success
- One of the error codes from SDMMC/SPI host controller

esp_err_t **essl_wait_int** (*essl_handle_t* handle, uint32_t wait_ms)

wait for an interrupt of the slave

Parameters
- **handle** –Handle of an ESSL device.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: If interrupt is triggered.
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- ESP_ERR_TIMEOUT: No interrupts before timeout.

esp_err_t **essl_clear_intr** (*essl_handle_t* handle, uint32_t intr_mask, uint32_t wait_ms)

Clear interrupt bits of ESSL slave. All the bits set in the mask will be cleared, while other bits will stay the same.

Parameters
- **handle** –Handle of an ESSL device.
- **intr_mask** –Mask of interrupt bits to clear.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

esp_err_t **essl_get_intr** (*essl_handle_t* handle, uint32_t *intr_raw, uint32_t *intr_st, uint32_t wait_ms)

Get interrupt bits of ESSL slave.

Parameters
- **handle** –Handle of an ESSL device.

- **intr_raw** –Output of the raw interrupt bits. Set to NULL if only masked bits are read.
- **intr_st** –Output of the masked interrupt bits. set to NULL if only raw bits are read.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns

- ESP_OK: Success
- ESP_INVALID_ARG: If both `intr_raw` and `intr_st` are NULL.
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

esp_err_t **essl_set_intr_ena**(*essl_handle_t* handle, uint32_t ena_mask, uint32_t wait_ms)

Set interrupt enable bits of ESSL slave. The slave only sends interrupt on the line when there is a bit both the raw status and the enable are set.

Parameters

- **handle** –Handle of an ESSL device.
- **ena_mask** –Mask of the interrupt bits to enable.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns

- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

esp_err_t **essl_get_intr_ena**(*essl_handle_t* handle, uint32_t *ena_mask_o, uint32_t wait_ms)

Get interrupt enable bits of ESSL slave.

Parameters

- **handle** –Handle of an ESSL device.
- **ena_mask_o** –Output of interrupt bit enable mask.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns

- ESP_OK Success
- One of the error codes from SDMMC host controller

esp_err_t **essl_send_slave_intr**(*essl_handle_t* handle, uint32_t intr_mask, uint32_t wait_ms)

Send interrupts to slave. Each bit of the interrupt will be triggered.

Parameters

- **handle** –Handle of an ESSL device.
- **intr_mask** –Mask of interrupt bits to send to slave.
- **wait_ms** –Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns

- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

Type Definitions

typedef struct essl_dev_t ***essl_handle_t**

Handle of an ESSL device.

Header File

- components/driver/test/esp_serial_slave_link/include/esp_serial_slave_link/essl_sdio.h

Functions

esp_err_t **essl_sdio_init_dev**(*essl_handle_t* *out_handle, const *essl_sdio_config_t* *config)

Initialize the ESSL SDIO device and get its handle.

Parameters

- **out_handle** –Output of the handle.
- **config** –Configuration for the ESSL SDIO device.

Returns

- ESP_OK: on success
- ESP_ERR_NO_MEM: memory exhausted.

esp_err_t **essl_sdio_deinit_dev**(*essl_handle_t* handle)

Deinitialize and free the space used by the ESSL SDIO device.

Parameters **handle** –Handle of the ESSL SDIO device to deinit.

Returns

- ESP_OK: on success
- ESP_ERR_INVALID_ARG: wrong handle passed

Structures

struct **essl_sdio_config_t**

Configuration for the ESSL SDIO device.

Public Members

sdmmc_card_t ***card**

The initialized sdmmc card pointer of the slave.

int **recv_buffer_size**

The pre-negotiated recv buffer size used by both the host and the slave.

Header File

- components/driver/test/esp_serial_slave_link/include/esp_serial_slave_link/essl_spi.h

Functions

esp_err_t **essl_spi_init_dev**(*essl_handle_t* *out_handle, const *essl_spi_config_t* *init_config)

Initialize the ESSL SPI device function list and get its handle.

Parameters

- **out_handle** –**[out]** Output of the handle
- **init_config** –Configuration for the ESSL SPI device

Returns

- ESP_OK: On success
- ESP_ERR_NO_MEM: Memory exhausted
- ESP_ERR_INVALID_STATE: SPI driver is not initialized
- ESP_ERR_INVALID_ARG: Wrong register ID

esp_err_t **essl_spi_deinit_dev**(*essl_handle_t* handle)

Deinitialize the ESSL SPI device and free the memory used by the device.

Parameters **handle** –Handle of the ESSL SPI device

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_STATE: ESSL SPI is not in use

esp_err_t **essl_spi_read_reg**(void *arg, uint8_t addr, uint8_t *out_value, uint32_t wait_ms)

Read from the shared registers.

Note: The registers for Master/Slave synchronization are reserved. Do not use them. (see `rx_sync_reg` in *essl_spi_config_t*)

Parameters

- **arg** –Context of the component. (Member `arg` from `essl_handle_t`)
- **addr** –Address of the shared registers. (Valid: 0 ~ SOC_SPI_MAXIMUM_BUFFER_SIZE, registers for M/S sync are reserved, see note1).
- **out_value** –**[out]** Read buffer for the shared registers.
- **wait_ms** –Time to wait before timeout (reserved for future use, user should set this to 0).

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: ESSL SPI has not been initialized.
- ESP_ERR_INVALID_ARG: The address argument is not valid. See note 1.
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_get_packet**(void *arg, void *out_data, size_t size, uint32_t wait_ms)

Get a packet from Slave.

Parameters

- **arg** –Context of the component. (Member `arg` from `essl_handle_t`)
- **out_data** –**[out]** Output data address
- **size** –The size of the output data.
- **wait_ms** –Time to wait before timeout (reserved for future use, user should set this to 0).

Returns

- ESP_OK: On Success
- ESP_ERR_INVALID_STATE: ESSL SPI has not been initialized.
- ESP_ERR_INVALID_ARG: The output data address is neither DMA capable nor 4 byte-aligned
- ESP_ERR_INVALID_SIZE: Master requires `size` bytes of data but Slave did not load enough bytes.

esp_err_t **essl_spi_write_reg**(void *arg, uint8_t addr, uint8_t value, uint8_t *out_value, uint32_t wait_ms)

Write to the shared registers.

Note: The registers for Master/Slave synchronization are reserved. Do not use them. (see `tx_sync_reg` in *essl_spi_config_t*)

Note: Feature of checking the actual written value (`out_value`) is not supported.

Parameters

- **arg** –Context of the component. (Member `arg` from `essl_handle_t`)
- **addr** –Address of the shared registers. (Valid: 0 ~ SOC_SPI_MAXIMUM_BUFFER_SIZE, registers for M/S sync are reserved, see note1)
- **value** –Buffer for data to send, should be align to 4.
- **out_value** –**[out]** Not supported, should be set to NULL.
- **wait_ms** –Time to wait before timeout (reserved for future use, user should set this to 0).

Returns

- ESP_OK: success

- ESP_ERR_INVALID_STATE: ESSL SPI has not been initialized.
- ESP_ERR_INVALID_ARG: The address argument is not valid. See note 1.
- ESP_ERR_NOT_SUPPORTED: Should set `out_value` to NULL. See note 2.
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_send_packet**(void *arg, const void *data, size_t size, uint32_t wait_ms)

Send a packet to Slave.

Parameters

- **arg** –Context of the component. (Member `arg` from `essl_handle_t`)
- **data** –Address of the data to send
- **size** –Size of the data to send.
- **wait_ms** –Time to wait before timeout (reserved for future use, user should set this to 0).

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_STATE: ESSL SPI has not been initialized.
- ESP_ERR_INVALID_ARG: The data address is not DMA capable
- ESP_ERR_INVALID_SIZE: Master will send `size` bytes of data but Slave did not load enough RX buffer

void **essl_spi_reset_cnt**(void *arg)

Reset the counter in Master context.

Note: Shall only be called if the slave has reset its counter. Else, Slave and Master would be desynchronized

---

Parameters **arg** –Context of the component. (Member `arg` from `essl_handle_t`)

esp_err_t **essl_spi_rdbuf**(*spi_device_handle_t* spi, uint8_t *out_data, int addr, int len, uint32_t flags)

Read the shared buffer from the slave in ISR way.

Note: The slave's HW doesn't guarantee the data in one SPI transaction is consistent. It sends data in unit of byte. In other words, if the slave SW attempts to update the shared register when a rdbuf SPI transaction is in-flight, the data got by the master will be the combination of bytes of different writes of slave SW.

Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

Parameters

- **spi** –SPI device handle representing the slave
- **out_data** –**[out]** Buffer for read data, strongly suggested to be in the DRAM and aligned to 4
- **addr** –Address of the slave shared buffer
- **len** –Length to read
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: on success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_rdbuf_polling**(*spi_device_handle_t* spi, uint8_t *out_data, int addr, int len, uint32_t flags)

Read the shared buffer from the slave in polling way.

Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

Parameters

- **`spi`** –SPI device handle representing the slave
- **`out_data`** –**[out]** Buffer for read data, strongly suggested to be in the DRAM and aligned to 4
- **`addr`** –Address of the slave shared buffer
- **`len`** –Length to read
- **`flags`** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: on success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **`essl_spi_wrbuf`**(*spi_device_handle_t* spi, const uint8_t *data, int addr, int len, uint32_t flags)

Write the shared buffer of the slave in ISR way.

Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

Parameters

- **`spi`** –SPI device handle representing the slave
- **`data`** –Buffer for data to send, strongly suggested to be in the DRAM
- **`addr`** –Address of the slave shared buffer,
- **`len`** –Length to write
- **`flags`** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **`essl_spi_wrbuf_polling`**(*spi_device_handle_t* spi, const uint8_t *data, int addr, int len, uint32_t flags)

Write the shared buffer of the slave in polling way.

Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

Parameters

- **`spi`** –SPI device handle representing the slave
- **`data`** –Buffer for data to send, strongly suggested to be in the DRAM
- **`addr`** –Address of the slave shared buffer,
- **`len`** –Length to write
- **`flags`** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_polling_transmit`.

esp_err_t **`essl_spi_rddma`**(*spi_device_handle_t* spi, uint8_t *out_data, int len, int seg_len, uint32_t flags)

Receive long buffer in segments from the slave through its DMA.

Note: This function combines several :cpp:func:`essl_spi_rddma_seg` and one :cpp:func:`essl_spi_rddma_done` at the end. Used when the slave is working in segment mode.

Parameters

- **spi** –SPI device handle representing the slave
- **out_data** –**[out]** Buffer to hold the received data, strongly suggested to be in the DRAM and aligned to 4
- **len** –Total length of data to receive.
- **seg_len** –Length of each segment, which is not larger than the maximum transaction length allowed for the spi device. Suggested to be multiples of 4. When set < 0, means send all data in one segment (the `rddma_done` will still be sent.)
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_rddma_seg**(*spi_device_handle_t* spi, uint8_t *out_data, int seg_len, uint32_t flags)

Read one data segment from the slave through its DMA.

Note: To read long buffer, call :cpp:func:`essl_spi_rddma` instead.

Parameters

- **spi** –SPI device handle representing the slave
- **out_data** –**[out]** Buffer to hold the received data. strongly suggested to be in the DRAM and aligned to 4
- **seg_len** –Length of this segment
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_rddma_done**(*spi_device_handle_t* spi, uint32_t flags)

Send the `rddma_done` command to the slave. Upon receiving this command, the slave will stop sending the current buffer even there are data unsent, and maybe prepare the next buffer to send.

Note: This is required only when the slave is working in segment mode.

Parameters

- **spi** –SPI device handle representing the slave
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_wrdma**(*spi_device_handle_t* spi, const uint8_t *data, int len, int seg_len, uint32_t flags)

Send long buffer in segments to the slave through its DMA.

Note: This function combines several :cpp:func:`essl_spi_wrdma_seg` and one :cpp:func:`essl_spi_wrdma_done` at the end. Used when the slave is working in segment mode.

Parameters

- **spi** –SPI device handle representing the slave
- **data** –Buffer for data to send, strongly suggested to be in the DRAM
- **len** –Total length of data to send.
- **seg_len** –Length of each segment, which is not larger than the maximum transaction length allowed for the spi device. Suggested to be multiples of 4. When set < 0, means send all data in one segment (the `wrdma_done` will still be sent.)
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_wrdma_seg**(*spi_device_handle_t* spi, const uint8_t *data, int seg_len, uint32_t flags)

Send one data segment to the slave through its DMA.

Note: To send long buffer, call :cpp:func:`essl_spi_wrdma` instead.

Parameters

- **spi** –SPI device handle representing the slave
- **data** –Buffer for data to send, strongly suggested to be in the DRAM
- **seg_len** –Length of this segment
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

esp_err_t **essl_spi_wrdma_done**(*spi_device_handle_t* spi, uint32_t flags)

Send the `wrdma_done` command to the slave. Upon receiving this command, the slave will stop receiving, process the received data, and maybe prepare the next buffer to receive.

Note: This is required only when the slave is working in segment mode.

Parameters

- **spi** –SPI device handle representing the slave
- **flags** –`SPI_TRANS_*` flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

Structures

struct **essl_spi_config_t**

Configuration of ESSL SPI device.

Public Members

spi_device_handle_t ***spi**

Pointer to SPI device handle.

uint32_t **tx_buf_size**

The pre-negotiated Master TX buffer size used by both the host and the slave.

uint8_t **tx_sync_reg**

The pre-negotiated register ID for Master-TX-SLAVE-RX synchronization. 1 word (4 Bytes) will be reserved for the synchronization.

uint8_t **rx_sync_reg**

The pre-negotiated register ID for Master-RX-Slave-TX synchronization. 1 word (4 Bytes) will be reserved for the synchronization.

2.2.8 ESP x509 Certificate Bundle

Overview

The ESP x509 Certificate Bundle API provides an easy way to include a bundle of custom x509 root certificates for TLS server verification.

Note: The bundle is currently not available when using WolfSSL.

The bundle comes with the complete list of root certificates from Mozilla's NSS root certificate store. Using the gen_crt_bundle.py python utility the certificates' subject name and public key are stored in a file and embedded in the ESP32 binary.

When generating the bundle you may choose between:

- The full root certificate bundle from Mozilla, containing more than 130 certificates. The current bundle was updated Tue Jul 19 03:12:06 2022 GMT.
- A pre-selected filter list of the name of the most commonly used root certificates, reducing the amount of certificates to around 35 while still having around 90 % coverage according to market share statistics.

In addition it is possible to specify a path to a certificate file or a directory containing certificates which then will be added to the generated bundle.

Note: Trusting all root certificates means the list will have to be updated if any of the certificates are retracted. This includes removing them from *cacrt_all.pem*.

Configuration

Most configuration is done through menuconfig. CMake will generate the bundle according to the configuration and embed it.

- *CONFIG_MBEDTLS_CERTIFICATE_BUNDLE*: automatically build and attach the bundle.
- *CONFIG_MBEDTLS_DEFAULT_CERTIFICATE_BUNDLE*: decide which certificates to include from the complete root list.
- *CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE_PATH*: specify the path of any additional certificates to embed in the bundle.

To enable the bundle when using ESP-TLS simply pass the function pointer to the bundle attach function:

```
esp_tls_cfg_t cfg = {
    .crt_bundle_attach = esp_crt_bundle_attach,
};
```

This is done to avoid embedding the certificate bundle unless activated by the user.

If using mbedTLS directly then the bundle may be activated by directly calling the attach function during the setup process:

```
mbedtls_ssl_config conf;
mbedtls_ssl_config_init(&conf);

esp_crt_bundle_attach(&conf);
```

Generating the List of Root Certificates

The list of root certificates comes from Mozilla's NSS root certificate store, which can be found here The list can be downloaded and created by running the script `mk-ca-bundle.pl` that is distributed as a part of curl. Another alternative would be to download the finished list directly from the curl website: CA certificates extracted from Mozilla

The common certificates bundle were made by selecting the authorities with a market share of more than 1 % from w3tech's SSL Survey. These authorities were then used to pick the names of the certificates for the filter list, *cmn_crt_authorities.csv*, from this list provided by Mozilla.

Updating the Certificate Bundle

The bundle is embedded into the app and can be updated along with the app by an OTA update. If you want to include a more up-to-date bundle than the bundle currently included in ESP-IDF, then the certificate list can be downloaded from Mozilla as described in *Generating the List of Root Certificates*.

Application Example

Simple HTTPS example that uses ESP-TLS to establish a secure socket connection using the certificate bundle with two custom certificates added for verification: protocols/https_x509_bundle.

HTTPS example that uses ESP-TLS and the default bundle: protocols/https_request.

HTTPS example that uses mbedTLS and the default bundle: protocols/https_mbedtls.

API Reference

Header File

- components/mbedtls/esp_crt_bundle/include/esp_crt_bundle.h

Functions

esp_err_t **esp_crt_bundle_attach**(void *conf)

Attach and enable use of a bundle for certificate verification.

Attach and enable use of a bundle for certificate verification through a verification callback. If no specific bundle has been set through esp_crt_bundle_set() it will default to the bundle defined in menuconfig and embedded in the binary.

Parameters **conf** –**[in]** The config struct for the SSL connection.

Returns

- ESP_OK if adding certificates was successful.
- Other if an error occured or an action must be taken by the calling process.

void **esp_crt_bundle_detach**(mbedtls_ssl_config *conf)

Disable and dealloc the certification bundle.

Removes the certificate verification callback and deallocates used resources

Parameters **conf** –**[in]** The config struct for the SSL connection.

esp_err_t **esp_crt_bundle_set** (const uint8_t *x509_bundle, size_t bundle_size)

Set the default certificate bundle used for verification.

Overrides the default certificate bundle only in case of successful initialization. In most use cases the bundle should be set through menuconfig. The bundle needs to be sorted by subject name since binary search is used to find certificates.

Parameters

- **x509_bundle** –**[in]** A pointer to the certificate bundle.
- **bundle_size** –**[in]** Size of the certificate bundle in bytes.

Returns

- ESP_OK if adding certificates was successful.
- Other if an error occured or an action must be taken by the calling process.

2.2.9 HTTP Server

Overview

The HTTP Server component provides an ability for running a lightweight web server on ESP32. Following are detailed steps to use the API exposed by HTTP Server:

- *httpd_start()*: Creates an instance of HTTP server, allocate memory/resources for it depending upon the specified configuration and outputs a handle to the server instance. The server has both, a listening socket (TCP) for HTTP traffic, and a control socket (UDP) for control signals, which are selected in a round robin fashion in the server task loop. The task priority and stack size are configurable during server instance creation by passing httpd_config_t structure to httpd_start(). TCP traffic is parsed as HTTP requests and, depending on the requested URI, user registered handlers are invoked which are supposed to send back HTTP response packets.
- *httpd_stop()*: This stops the server with the provided handle and frees up any associated memory/resources. This is a blocking function that first signals a halt to the server task and then waits for the task to terminate. While stopping, the task will close all open connections, remove registered URI handlers and reset all session context data to empty.
- *httpd_register_uri_handler()*: A URI handler is registered by passing object of type `httpd_uri_t` structure which has members including `uri` name, `method` type (eg. `HTTPD_GET/HTTPD_POST/HTTPD_PUT` etc.), function pointer of type `esp_err_t *handler (httpd_req_t *req)` and `user_ctx` pointer to user context data.

Application Example

```
/* Our URI handler function to be called during GET /uri request */
esp_err_t get_handler(httpd_req_t *req)
{
    /* Send a simple response */
    const char resp[] = "URI GET Response";
    httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
    return ESP_OK;
}

/* Our URI handler function to be called during POST /uri request */
esp_err_t post_handler(httpd_req_t *req)
{
    /* Destination buffer for content of HTTP POST request.
     * httpd_req_recv() accepts char* only, but content could
     * as well be any binary data (needs type casting).
     * In case of string data, null termination will be absent, and
```

(continues on next page)

(continued from previous page)

```
     * content length would give length of string */
    char content[100];

    /* Truncate if content length larger than the buffer */
    size_t recv_size = MIN(req->content_len, sizeof(content));

    int ret = httpd_req_recv(req, content, recv_size);
    if (ret <= 0) {  /* 0 return value indicates connection closed */
        /* Check if timeout occurred */
        if (ret == HTTPD_SOCK_ERR_TIMEOUT) {
            /* In case of timeout one can choose to retry calling
             * httpd_req_recv(), but to keep it simple, here we
             * respond with an HTTP 408 (Request Timeout) error */
            httpd_resp_send_408(req);
        }
        /* In case of error, returning ESP_FAIL will
         * ensure that the underlying socket is closed */
        return ESP_FAIL;
    }

    /* Send a simple response */
    const char resp[] = "URI POST Response";
    httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
    return ESP_OK;
}

/* URI handler structure for GET /uri */
httpd_uri_t uri_get = {
    .uri      = "/uri",
    .method   = HTTP_GET,
    .handler  = get_handler,
    .user_ctx = NULL
};

/* URI handler structure for POST /uri */
httpd_uri_t uri_post = {
    .uri      = "/uri",
    .method   = HTTP_POST,
    .handler  = post_handler,
    .user_ctx = NULL
};

/* Function for starting the webserver */
httpd_handle_t start_webserver(void)
{
    /* Generate default configuration */
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    /* Empty handle to esp_http_server */
    httpd_handle_t server = NULL;

    /* Start the httpd server */
    if (httpd_start(&server, &config) == ESP_OK) {
        /* Register URI handlers */
        httpd_register_uri_handler(server, &uri_get);
        httpd_register_uri_handler(server, &uri_post);
    }
    /* If server failed to start, handle will be NULL */
    return server;
}
```

(continues on next page)

(continued from previous page)

```
/* Function for stopping the webserver */
void stop_webserver(httpd_handle_t server)
{
    if (server) {
        /* Stop the httpd server */
        httpd_stop(server);
    }
}
```

Simple HTTP Server Example Check HTTP server example under protocols/http_server/simple where handling of arbitrary content lengths, reading request headers and URL query parameters, and setting response headers is demonstrated.

Persistent Connections

HTTP server features persistent connections, allowing for the re-use of the same connection (session) for several transfers, all the while maintaining context specific data for the session. Context data may be allocated dynamically by the handler in which case a custom function may need to be specified for freeing this data when the connection/session is closed.

Persistent Connections Example

```
/* Custom function to free context */
void free_ctx_func(void *ctx)
{
    /* Could be something other than free */
    free(ctx);
}

esp_err_t adder_post_handler(httpd_req_t *req)
{
    /* Create session's context if not already available */
    if (! req->sess_ctx) {
        req->sess_ctx = malloc(sizeof(ANY_DATA_TYPE));   /*!< Pointer to context␣
↪data */
        req->free_ctx = free_ctx_func;                   /*!< Function to free␣
↪context data */
    }

    /* Access context data */
    ANY_DATA_TYPE *ctx_data = (ANY_DATA_TYPE *)req->sess_ctx;

    /* Respond */
    ................
    ................
    ................

    return ESP_OK;
}
```

Check the example under protocols/http_server/persistent_sockets.

Websocket Server

The HTTP server component provides websocket support. The websocket feature can be enabled in menuconfig using the *CONFIG_HTTPD_WS_SUPPORT* option. Please refer to the protocols/http_server/ws_echo_server example which demonstrates usage of the websocket feature.

Event Handling

ESP HTTP Server has various events for which a handler can be triggered by *the Event Loop library* when the particular event occurs. The handler has to be registered using `esp_event_handler_register()`. This helps in event handling for ESP HTTP Server. `esp_http_server_event_id_t` has all the events which can happen for ESP HTTP Server.

Expected data type for different ESP HTTP Server events in event loop:

- HTTP_SERVER_EVENT_ERROR : `httpd_err_code_t`
- HTTP_SERVER_EVENT_START : `NULL`
- HTTP_SERVER_EVENT_ON_CONNECTED : `int`
- HTTP_SERVER_EVENT_ON_HEADER : `int`
- HTTP_SERVER_EVENT_HEADERS_SENT : `int`
- HTTP_SERVER_EVENT_ON_DATA : `esp_http_server_event_data`
- HTTP_SERVER_EVENT_SENT_DATA : `esp_http_server_event_data`
- HTTP_SERVER_EVENT_DISCONNECTED : `int`
- HTTP_SERVER_EVENT_STOP : `NULL`

API Reference

Header File

- components/esp_http_server/include/esp_http_server.h

Functions

esp_err_t **httpd_register_uri_handler** (*httpd_handle_t* handle, const *httpd_uri_t* *uri_handler)

Registers a URI handler.

Example usage:

```
esp_err_t my_uri_handler(httpd_req_t* req)
{
    // Recv , Process and Send
    ....
    ....
    ....

    // Fail condition
    if (....) {
        // Return fail to close session //
        return ESP_FAIL;
    }

    // On success
    return ESP_OK;
}

// URI handler structure
httpd_uri_t my_uri {
    .uri      = "/my_uri/path/xyz",
    .method   = HTTPD_GET,
    .handler  = my_uri_handler,
    .user_ctx = NULL
};

// Register handler
if (httpd_register_uri_handler(server_handle, &my_uri) != ESP_OK) {
   // If failed to register handler
```

(continues on next page)

(continued from previous page)

```
    ....
}
```

Note: URI handlers can be registered in real time as long as the server handle is valid.

Parameters

- **handle** –**[in]** handle to HTTPD server instance
- **uri_handler** –**[in]** pointer to handler that needs to be registered

Returns

- ESP_OK : On successfully registering the handler
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_HANDLERS_FULL : If no slots left for new handler
- ESP_ERR_HTTPD_HANDLER_EXISTS : If handler with same URI and method is already registered

esp_err_t **httpd_unregister_uri_handler** (*httpd_handle_t* handle, const char *uri, *httpd_method_t* method)

Unregister a URI handler.

Parameters

- **handle** –**[in]** handle to HTTPD server instance
- **uri** –**[in]** URI string
- **method** –**[in]** HTTP method

Returns

- ESP_OK : On successfully deregistering the handler
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_FOUND : Handler with specified URI and method not found

esp_err_t **httpd_unregister_uri** (*httpd_handle_t* handle, const char *uri)

Unregister all URI handlers with the specified uri string.

Parameters

- **handle** –**[in]** handle to HTTPD server instance
- **uri** –**[in]** uri string specifying all handlers that need to be deregisterd

Returns

- ESP_OK : On successfully deregistering all such handlers
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_FOUND : No handler registered with specified uri string

esp_err_t **httpd_sess_set_recv_override** (*httpd_handle_t* hd, int sockfd, *httpd_recv_func_t* recv_func)

Override web server’ s receive function (by session FD)

This function overrides the web server’ s receive function. This same function is used to read HTTP request packets.

Note: This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using httpd_req_to_sockfd()

Parameters

- **hd** –**[in]** HTTPD instance handle
- **sockfd** –**[in]** Session socket FD
- **recv_func** –**[in]** The receive function to be set for this session

Returns

- ESP_OK : On successfully registering override

- ESP_ERR_INVALID_ARG : Null arguments

esp_err_t **httpd_sess_set_send_override** (*httpd_handle_t* hd, int sockfd, *httpd_send_func_t* send_func)

Override web server' s send function (by session FD)

This function overrides the web server' s send function. This same function is used to send out any response to any HTTP request.

Note: This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using httpd_req_to_sockfd()

Parameters

- **hd** –**[in]** HTTPD instance handle
- **sockfd** –**[in]** Session socket FD
- **send_func** –**[in]** The send function to be set for this session

Returns

- ESP_OK : On successfully registering override
- ESP_ERR_INVALID_ARG : Null arguments

esp_err_t **httpd_sess_set_pending_override** (*httpd_handle_t* hd, int sockfd, *httpd_pending_func_t* pending_func)

Override web server' s pending function (by session FD)

This function overrides the web server' s pending function. This function is used to test for pending bytes in a socket.

Note: This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using httpd_req_to_sockfd()

Parameters

- **hd** –**[in]** HTTPD instance handle
- **sockfd** –**[in]** Session socket FD
- **pending_func** –**[in]** The receive function to be set for this session

Returns

- ESP_OK : On successfully registering override
- ESP_ERR_INVALID_ARG : Null arguments

int **httpd_req_to_sockfd** (*httpd_req_t* *r)

Get the Socket Descriptor from the HTTP request.

This API will return the socket descriptor of the session for which URI handler was executed on reception of HTTP request. This is useful when user wants to call functions that require session socket fd, from within a URI handler, ie. : httpd_sess_get_ctx(), httpd_sess_trigger_close(), httpd_sess_update_lru_counter().

Note: This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.

Parameters **r** –**[in]** The request whose socket descriptor should be found

Returns

- Socket descriptor : The socket descriptor for this request
- -1 : Invalid/NULL request pointer

int **httpd_req_recv** (*httpd_req_t* *r, char *buf, size_t buf_len)

API to read content data from the HTTP request.

This API will read HTTP content data from the HTTP request into provided buffer. Use content_len provided in httpd_req_t structure to know the length of data to be fetched. If content_len is too large for the buffer then user may have to make multiple calls to this function, each time fetching 'buf_len' number of bytes, while the pointer to content data is incremented internally by the same number.

Note:

- This API is supported to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- If an error is returned, the URI handler must further return an error. This will ensure that the erroneous socket is closed and cleaned up by the web server.
- Presently Chunked Encoding is not supported

Parameters

- **r** –**[in]** The request being responded to
- **buf** –**[in]** Pointer to a buffer that the data will be read into
- **buf_len** –**[in]** Length of the buffer

Returns

- Bytes : Number of bytes read into the buffer successfully
- 0 : Buffer length parameter is zero / connection closed by peer
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket recv()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket recv()

size_t **httpd_req_get_hdr_value_len** (*httpd_req_t* *r, const char *field)

Search for a field in request headers and return the string length of it's value.

Note:

- This API is supported to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once httpd_resp_send() API is called all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters

- **r** –**[in]** The request being responded to
- **field** –**[in]** The header field to be searched in the request

Returns

- Length : If field is found in the request URL
- Zero : Field not found / Invalid request / Null arguments

esp_err_t **httpd_req_get_hdr_value_str** (*httpd_req_t* *r, const char *field, char *val, size_t val_size)

Get the value string of a field from the request headers.

Note:

- This API is supported to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once httpd_resp_send() API is called all request headers are purged, so request headers need be copied into separate buffers if they are required later.
- If output size is greater than input, then the value is truncated, accompanied by truncation error as return value.
- Use httpd_req_get_hdr_value_len() to know the right buffer length

---

Parameters

- **r** –**[in]** The request being responded to
- **field** –**[in]** The field to be searched in the header
- **val** –**[out]** Pointer to the buffer into which the value will be copied if the field is found
- **val_size** –**[in]** Size of the user buffer "val"

Returns

- ESP_OK : Field found in the request header and value string copied
- ESP_ERR_NOT_FOUND : Key not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_INVALID_REQ : Invalid HTTP request pointer
- ESP_ERR_HTTPD_RESULT_TRUNC : Value string truncated

size_t **httpd_req_get_url_query_len**(*httpd_req_t* *r)

Get Query string length from the request URL.

Note: This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid

---

Parameters **r** –**[in]** The request being responded to

Returns

- Length : Query is found in the request URL
- Zero : Query not found / Null arguments / Invalid request

esp_err_t **httpd_req_get_url_query_str**(*httpd_req_t* *r, char *buf, size_t buf_len)

Get Query string from the request URL.

Note:

- Presently, the user can fetch the full URL query string, but decoding will have to be performed by the user. Request headers can be read using httpd_req_get_hdr_value_str() to know the 'Content-Type' (eg. Content-Type: application/x-www-form-urlencoded) and then the appropriate decoding algorithm needs to be applied.
- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid
- If output size is greater than input, then the value is truncated, accompanied by truncation error as return value
- Prior to calling this function, one can use httpd_req_get_url_query_len() to know the query string length beforehand and hence allocate the buffer of right size (usually query string length + 1 for null termination) for storing the query string

---

Parameters

- **r** –**[in]** The request being responded to
- **buf** –**[out]** Pointer to the buffer into which the query string will be copied (if found)
- **buf_len** –**[in]** Length of output buffer

Returns

- ESP_OK : Query is found in the request URL and copied to buffer
- ESP_ERR_NOT_FOUND : Query not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_INVALID_REQ : Invalid HTTP request pointer
- ESP_ERR_HTTPD_RESULT_TRUNC : Query string truncated

esp_err_t **httpd_query_key_value**(const char *qry, const char *key, char *val, size_t val_size)

Helper function to get a URL query tag from a query string of the type param1=val1&param2=val2.

Note:

- The components of URL query string (keys and values) are not URLdecoded. The user must check for 'Content-Type' field in the request headers and then depending upon the specified encoding (URLencoded or otherwise) apply the appropriate decoding algorithm.
- If actual value size is greater than val_size, then the value is truncated, accompanied by truncation error as return value.

Parameters

- **qry** –**[in]** Pointer to query string
- **key** –**[in]** The key to be searched in the query string
- **val** –**[out]** Pointer to the buffer into which the value will be copied if the key is found
- **val_size** –**[in]** Size of the user buffer "val"

Returns

- ESP_OK : Key is found in the URL query string and copied to buffer
- ESP_ERR_NOT_FOUND : Key not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESULT_TRUNC : Value string truncated

esp_err_t **httpd_req_get_cookie_val**(*httpd_req_t* *req, const char *cookie_name, char *val, size_t *val_size)

Get the value string of a cookie value from the "Cookie" request headers by cookie name.

Parameters

- **req** –**[in]** Pointer to the HTTP request
- **cookie_name** –**[in]** The cookie name to be searched in the request
- **val** –**[out]** Pointer to the buffer into which the value of cookie will be copied if the cookie is found
- **val_size** –**[inout]** Pointer to size of the user buffer "val". This variable will contain cookie length if ESP_OK is returned and required buffer length incase ESP_ERR_HTTPD_RESULT_TRUNC is returned.

Returns

- ESP_OK : Key is found in the cookie string and copied to buffer
- ESP_ERR_NOT_FOUND : Key not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESULT_TRUNC : Value string truncated
- ESP_ERR_NO_MEM : Memory allocation failure

bool **httpd_uri_match_wildcard**(const char *uri_template, const char *uri_to_match, size_t match_upto)

Test if a URI matches the given wildcard template.

Template may end with "?" to make the previous character optional (typically a slash), "*" for a wildcard match, and "?*" to make the previous character optional, and if present, allow anything to follow.

Example:

- * matches everything
- /foo/? matches /foo and /foo/
- /foo/* (sans the backslash) matches /foo/ and /foo/bar, but not /foo or /fo
- /foo/?* or /foo/*? (sans the backslash) matches /foo/, /foo/bar, and also /foo, but not /foox or /fo

The special characters "?" and "*" anywhere else in the template will be taken literally.

Parameters

- **uri_template** –**[in]** URI template (pattern)
- **uri_to_match** –**[in]** URI to be matched

- **match_upto** –**[in]** how many characters of the URI buffer to test (there may be trailing query string etc.)

Returns true if a match was found

esp_err_t **httpd_resp_send** (*httpd_req_t* *r, const char *buf, ssize_t buf_len)

API to send a complete HTTP response.

This API will send the data as an HTTP response to the request. This assumes that you have the entire response ready in a single buffer. If you wish to send response in incremental chunks use httpd_resp_send_chunk() instead.

If no status code and content-type were set, by default this will send 200 OK status code and content type as text/html. You may call the following functions before this API to configure the response headers : httpd_resp_set_status() - for setting the HTTP status string, httpd_resp_set_type() - for setting the Content Type, httpd_resp_set_hdr() - for appending any additional field value entries in the response header

---

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once this API is called, the request has been responded to.
- No additional data can then be sent for the request.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters

- **r** –**[in]** The request being responded to
- **buf** –**[in]** Buffer from where the content is to be fetched
- **buf_len** –**[in]** Length of the buffer, HTTPD_RESP_USE_STRLEN to use strlen()

Returns

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null request pointer
- ESP_ERR_HTTPD_RESP_HDR : Essential headers are too large for internal buffer
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request

esp_err_t **httpd_resp_send_chunk** (*httpd_req_t* *r, const char *buf, ssize_t buf_len)

API to send one HTTP chunk.

This API will send the data as an HTTP response to the request. This API will use chunked-encoding and send the response in the form of chunks. If you have the entire response contained in a single buffer, please use httpd_resp_send() instead.

If no status code and content-type were set, by default this will send 200 OK status code and content type as text/html. You may call the following functions before this API to configure the response headers httpd_resp_set_status() - for setting the HTTP status string, httpd_resp_set_type() - for setting the Content Type, httpd_resp_set_hdr() - for appending any additional field value entries in the response header

---

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- When you are finished sending all your chunks, you must call this function with buf_len as 0.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters

- **r** –**[in]** The request being responded to

- **buf** –**[in]** Pointer to a buffer that stores the data
- **buf_len** –**[in]** Length of the buffer, HTTPD_RESP_USE_STRLEN to use strlen()

Returns

- ESP_OK : On successfully sending the response packet chunk
- ESP_ERR_INVALID_ARG : Null request pointer
- ESP_ERR_HTTPD_RESP_HDR : Essential headers are too large for internal buffer
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

static inline *esp_err_t* **httpd_resp_sendstr**(*httpd_req_t* *r, const char *str)

API to send a complete string as HTTP response.

This API simply calls http_resp_send with buffer length set to string length assuming the buffer contains a null terminated string

Parameters

- **r** –**[in]** The request being responded to
- **str** –**[in]** String to be sent as response body

Returns

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null request pointer
- ESP_ERR_HTTPD_RESP_HDR : Essential headers are too large for internal buffer
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request

static inline *esp_err_t* **httpd_resp_sendstr_chunk**(*httpd_req_t* *r, const char *str)

API to send a string as an HTTP response chunk.

This API simply calls http_resp_send_chunk with buffer length set to string length assuming the buffer contains a null terminated string

Parameters

- **r** –**[in]** The request being responded to
- **str** –**[in]** String to be sent as response body (NULL to finish response packet)

Returns

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null request pointer
- ESP_ERR_HTTPD_RESP_HDR : Essential headers are too large for internal buffer
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request

esp_err_t **httpd_resp_set_status**(*httpd_req_t* *r, const char *status)

API to set the HTTP status code.

This API sets the status of the HTTP response to the value specified. By default, the '200 OK' response is sent as the response.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- This API only sets the status to this value. The status isn't sent out until any of the send APIs is executed.
- Make sure that the lifetime of the status string is valid till send function is called.

Parameters

- **r** –**[in]** The request being responded to
- **status** –**[in]** The HTTP status code of this response

Returns

- ESP_OK : On success
- ESP_ERR_INVALID_ARG : Null arguments

- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

esp_err_t **httpd_resp_set_type** (*httpd_req_t* *r, const char *type)

API to set the HTTP content type.

This API sets the ‘Content Type’ field of the response. The default content type is ‘text/html’.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- This API only sets the content type to this value. The type isn’t sent out until any of the send APIs is executed.
- Make sure that the lifetime of the type string is valid till send function is called.

Parameters

- **r** –**[in]** The request being responded to
- **type** –**[in]** The Content Type of the response

Returns

- ESP_OK : On success
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

esp_err_t **httpd_resp_set_hdr** (*httpd_req_t* *r, const char *field, const char *value)

API to append any additional headers.

This API sets any additional header fields that need to be sent in the response.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- The header isn’t sent out until any of the send APIs is executed.
- The maximum allowed number of additional headers is limited to value of max_resp_headers in config structure.
- Make sure that the lifetime of the field value strings are valid till send function is called.

Parameters

- **r** –**[in]** The request being responded to
- **field** –**[in]** The field name of the HTTP header
- **value** –**[in]** The value of this HTTP header

Returns

- ESP_OK : On successfully appending new header
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_HDR : Total additional headers exceed max allowed
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

esp_err_t **httpd_resp_send_err** (*httpd_req_t* *req, *httpd_err_code_t* error, const char *msg)

For sending out error code in response to HTTP request.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

- If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Parameters
- **req** –**[in]** Pointer to the HTTP request for which the response needs to be sent
- **error** –**[in]** Error type to send
- **msg** –**[in]** Error message string (pass NULL for default message)

Returns
- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

static inline *esp_err_t* **httpd_resp_send_404** (*httpd_req_t* *r)

Helper function for HTTP 404.

Send HTTP 404 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters **r** –**[in]** The request being responded to

Returns
- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

static inline *esp_err_t* **httpd_resp_send_408** (*httpd_req_t* *r)

Helper function for HTTP 408.

Send HTTP 408 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters **r** –**[in]** The request being responded to

Returns
- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

static inline *esp_err_t* **httpd_resp_send_500** (*httpd_req_t* *r)

Helper function for HTTP 500.

Send HTTP 500 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Parameters **r** –**[in]** The request being responded to

Returns

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

int **httpd_send** (*httpd_req_t* *r, const char *buf, size_t buf_len)

Raw HTTP send.

Call this API if you wish to construct your custom response packet. When using this, all essential header, eg. HTTP version, Status Code, Content Type and Length, Encoding, etc. will have to be constructed manually, and HTTP delimeters (CRLF) will need to be placed correctly for separating sub-sections of the HTTP response packet.

If the send override function is set, this API will end up calling that function eventually to send data out.

Note:

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Unless the response has the correct HTTP structure (which the user must now ensure) it is not guaranteed that it will be recognized by the client. For most cases, you wouldn't have to call this API, but you would rather use either of : httpd_resp_send(), httpd_resp_send_chunk()

Parameters

- **r** –**[in]** The request being responded to
- **buf** –**[in]** Buffer from where the fully constructed packet is to be read
- **buf_len** –**[in]** Length of the buffer

Returns

- Bytes : Number of bytes that were sent successfully
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket send()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket send()

int **httpd_socket_send** (*httpd_handle_t* hd, int sockfd, const char *buf, size_t buf_len, int flags)

A low level API to send data on a given socket

This internally calls the default send function, or the function registered by httpd_sess_set_send_override().

Note: This API is not recommended to be used in any request handler. Use this only for advanced use cases, wherein some asynchronous data is to be sent over a socket.

Parameters
- **hd** –**[in]** server instance
- **sockfd** –**[in]** session socket file descriptor
- **buf** –**[in]** buffer with bytes to send
- **buf_len** –**[in]** data size
- **flags** –**[in]** flags for the send() function

Returns
- Bytes : The number of bytes sent successfully
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket send()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket send()

int **httpd_socket_recv**(*httpd_handle_t* hd, int sockfd, char *buf, size_t buf_len, int flags)

A low level API to receive data from a given socket

This internally calls the default recv function, or the function registered by httpd_sess_set_recv_override().

Note: This API is not recommended to be used in any request handler. Use this only for advanced use cases, wherein some asynchronous communication is required.

Parameters
- **hd** –**[in]** server instance
- **sockfd** –**[in]** session socket file descriptor
- **buf** –**[in]** buffer with bytes to send
- **buf_len** –**[in]** data size
- **flags** –**[in]** flags for the send() function

Returns
- Bytes : The number of bytes received successfully
- 0 : Buffer length parameter is zero / connection closed by peer
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket recv()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket recv()

esp_err_t **httpd_register_err_handler**(*httpd_handle_t* handle, *httpd_err_code_t* error, *httpd_err_handler_func_t* handler_fn)

Function for registering HTTP error handlers.

This function maps a handler function to any supported error code given by `httpd_err_code_t`. See prototype `httpd_err_handler_func_t` above for details.

Parameters
- **handle** –**[in]** HTTP server handle
- **error** –**[in]** Error type
- **handler_fn** –**[in]** User implemented handler function (Pass NULL to unset any previously set handler)

Returns
- ESP_OK : handler registered successfully
- ESP_ERR_INVALID_ARG : invalid error code or server handle

esp_err_t **httpd_start**(*httpd_handle_t* *handle, const *httpd_config_t* *config)

Starts the web server.

Create an instance of HTTP server and allocate memory/resources for it depending upon the specified configuration.

Example usage:

```
//Function for starting the webserver
httpd_handle_t start_webserver(void)
{
    // Generate default configuration
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    // Empty handle to http_server
    httpd_handle_t server = NULL;

    // Start the httpd server
    if (httpd_start(&server, &config) == ESP_OK) {
        // Register URI handlers
        httpd_register_uri_handler(server, &uri_get);
        httpd_register_uri_handler(server, &uri_post);
    }
    // If server failed to start, handle will be NULL
    return server;
}
```

Parameters

- **config** –**[in]** Configuration for new instance of the server
- **handle** –**[out]** Handle to newly created instance of the server. NULL on error

Returns

- ESP_OK : Instance created successfully
- ESP_ERR_INVALID_ARG : Null argument(s)
- ESP_ERR_HTTPD_ALLOC_MEM : Failed to allocate memory for instance
- ESP_ERR_HTTPD_TASK : Failed to launch server task

esp_err_t **httpd_stop**(*httpd_handle_t* handle)

Stops the web server.

Deallocates memory/resources used by an HTTP server instance and deletes it. Once deleted the handle can no longer be used for accessing the instance.

Example usage:

```
// Function for stopping the webserver
void stop_webserver(httpd_handle_t server)
{
    // Ensure handle is non NULL
    if (server != NULL) {
        // Stop the httpd server
        httpd_stop(server);
    }
}
```

Parameters **handle** –**[in]** Handle to server returned by httpd_start

Returns

- ESP_OK : Server stopped successfully
- ESP_ERR_INVALID_ARG : Handle argument is Null

esp_err_t **httpd_queue_work**(*httpd_handle_t* handle, *httpd_work_fn_t* work, void *arg)

Queue execution of a function in HTTPD' s context.

This API queues a work function for asynchronous execution

Note: Some protocols require that the web server generate some asynchronous data and send it to the persistently opened connection. This facility is for use by such protocols.

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start
- **work** –**[in]** Pointer to the function to be executed in the HTTPD's context
- **arg** –**[in]** Pointer to the arguments that should be passed to this function

Returns

- ESP_OK : On successfully queueing the work
- ESP_FAIL : Failure in ctrl socket
- ESP_ERR_INVALID_ARG : Null arguments

void ***httpd_sess_get_ctx**(*httpd_handle_t* handle, int sockfd)

Get session context from socket descriptor.

Typically if a session context is created, it is available to URI handlers through the httpd_req_t structure. But, there are cases where the web server's send/receive functions may require the context (for example, for accessing keying information etc). Since the send/receive function only have the socket descriptor at their disposal, this API provides them with a way to retrieve the session context.

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start
- **sockfd** –**[in]** The socket descriptor for which the context should be extracted.

Returns

- void* : Pointer to the context associated with this session
- NULL : Empty context / Invalid handle / Invalid socket fd

void **httpd_sess_set_ctx**(*httpd_handle_t* handle, int sockfd, void *ctx, *httpd_free_ctx_fn_t* free_fn)

Set session context by socket descriptor.

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start
- **sockfd** –**[in]** The socket descriptor for which the context should be extracted.
- **ctx** –**[in]** Context object to assign to the session
- **free_fn** –**[in]** Function that should be called to free the context

void ***httpd_sess_get_transport_ctx**(*httpd_handle_t* handle, int sockfd)

Get session 'transport' context by socket descriptor.

This context is used by the send/receive functions, for example to manage SSL context.

See also:

httpd_sess_get_ctx()

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start
- **sockfd** –**[in]** The socket descriptor for which the context should be extracted.

Returns

- void* : Pointer to the transport context associated with this session
- NULL : Empty context / Invalid handle / Invalid socket fd

void **httpd_sess_set_transport_ctx**(*httpd_handle_t* handle, int sockfd, void *ctx, *httpd_free_ctx_fn_t* free_fn)

Set session 'transport' context by socket descriptor.

See also:

httpd_sess_set_ctx()

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start

- **sockfd** –**[in]** The socket descriptor for which the context should be extracted.
- **ctx** –**[in]** Transport context object to assign to the session
- **free_fn** –**[in]** Function that should be called to free the transport context

void ***httpd_get_global_user_ctx**(*httpd_handle_t* handle)

Get HTTPD global user context (it was set in the server config struct)

Parameters **handle** –**[in]** Handle to server returned by httpd_start
Returns global user context

void ***httpd_get_global_transport_ctx**(*httpd_handle_t* handle)

Get HTTPD global transport context (it was set in the server config struct)

Parameters **handle** –**[in]** Handle to server returned by httpd_start
Returns global transport context

esp_err_t **httpd_sess_trigger_close**(*httpd_handle_t* handle, int sockfd)

Trigger an httpd session close externally.

Note: Calling this API is only required in special circumstances wherein some application requires to close an httpd client session asynchronously.

Parameters
- **handle** –**[in]** Handle to server returned by httpd_start
- **sockfd** –**[in]** The socket descriptor of the session to be closed

Returns
- ESP_OK : On successfully initiating closure
- ESP_FAIL : Failure to queue work
- ESP_ERR_NOT_FOUND : Socket fd not found
- ESP_ERR_INVALID_ARG : Null arguments

esp_err_t **httpd_sess_update_lru_counter**(*httpd_handle_t* handle, int sockfd)

Update LRU counter for a given socket.

LRU Counters are internally associated with each session to monitor how recently a session exchanged traffic. When LRU purge is enabled, if a client is requesting for connection but maximum number of sockets/sessions is reached, then the session having the earliest LRU counter is closed automatically.

Updating the LRU counter manually prevents the socket from being purged due to the Least Recently Used (LRU) logic, even though it might not have received traffic for some time. This is useful when all open sockets/session are frequently exchanging traffic but the user specifically wants one of the sessions to be kept open, irrespective of when it last exchanged a packet.

Note: Calling this API is only necessary if the LRU Purge Enable option is enabled.

Parameters
- **handle** –**[in]** Handle to server returned by httpd_start
- **sockfd** –**[in]** The socket descriptor of the session for which LRU counter is to be updated

Returns
- ESP_OK : Socket found and LRU counter updated
- ESP_ERR_NOT_FOUND : Socket not found
- ESP_ERR_INVALID_ARG : Null arguments

esp_err_t **httpd_get_client_list**(*httpd_handle_t* handle, size_t *fds, int *client_fds)

Returns list of current socket descriptors of active sessions.

Note: Size of provided array has to be equal or greater then maximum number of opened sockets, configured upon initialization with max_open_sockets field in httpd_config_t structure.

Parameters

- **handle** –**[in]** Handle to server returned by httpd_start
- **fds** –**[inout]** In: Size of provided client_fds array Out: Number of valid client fds returned in client_fds,
- **client_fds** –**[out]** Array of client fds

Returns

- ESP_OK : Successfully retrieved session list
- ESP_ERR_INVALID_ARG : Wrong arguments or list is longer than provided array

Structures

struct **esp_http_server_event_data**

Argument structure for HTTP_SERVER_EVENT_ON_DATA and HTTP_SERVER_EVENT_SENT_DATA event

Public Members

int **fd**

Session socket file descriptor

int **data_len**

Data length

struct **httpd_config**

HTTP Server Configuration Structure.

Note: Use HTTPD_DEFAULT_CONFIG() to initialize the configuration to a default value and then modify only those fields that are specifically determined by the use case.

Public Members

unsigned **task_priority**

Priority of FreeRTOS task which runs the server

size_t **stack_size**

The maximum stack size allowed for the server task

BaseType_t **core_id**

The core the HTTP server task will run on

uint16_t **server_port**

TCP Port number for receiving and transmitting HTTP traffic

uint16_t **ctrl_port**

UDP Port number for asynchronously exchanging control signals between various components of the server

uint16_t **max_open_sockets**

Max number of sockets/clients connected at any time

uint16_t **max_uri_handlers**

Maximum allowed uri handlers

uint16_t **max_resp_headers**

Maximum allowed additional headers in HTTP response

uint16_t **backlog_conn**

Number of backlog connections

bool **lru_purge_enable**

Purge "Least Recently Used" connection

uint16_t **recv_wait_timeout**

Timeout for recv function (in seconds)

uint16_t **send_wait_timeout**

Timeout for send function (in seconds)

void ***global_user_ctx**

Global user context.

This field can be used to store arbitrary user data within the server context. The value can be retrieved using the server handle, available e.g. in the httpd_req_t struct.

When shutting down, the server frees up the user context by calling free() on the global_user_ctx field. If you wish to use a custom function for freeing the global user context, please specify that here.

httpd_free_ctx_fn_t **global_user_ctx_free_fn**

Free function for global user context

void ***global_transport_ctx**

Global transport context.

Similar to global_user_ctx, but used for session encoding or encryption (e.g. to hold the SSL context). It will be freed using free(), unless global_transport_ctx_free_fn is specified.

httpd_free_ctx_fn_t **global_transport_ctx_free_fn**

Free function for global transport context

bool **enable_so_linger**

bool to enable/disable linger

int **linger_timeout**

linger timeout (in seconds)

bool **keep_alive_enable**

Enable keep-alive timeout

int **keep_alive_idle**

Keep-alive idle time. Default is 5 (second)

int **keep_alive_interval**

Keep-alive interval time. Default is 5 (second)

int **keep_alive_count**

Keep-alive packet retry send count. Default is 3 counts

httpd_open_func_t **open_fn**

Custom session opening callback.

Called on a new session socket just after accept(), but before reading any data.

This is an opportunity to set up e.g. SSL encryption using global_transport_ctx and the send/recv/pending session overrides.

If a context needs to be maintained between these functions, store it in the session using httpd_sess_set_transport_ctx() and retrieve it later with httpd_sess_get_transport_ctx()

Returning a value other than ESP_OK will immediately close the new socket.

httpd_close_func_t **close_fn**

Custom session closing callback.

Called when a session is deleted, before freeing user and transport contexts and before closing the socket. This is a place for custom de-init code common to all sockets.

The server will only close the socket if no custom session closing callback is set. If a custom callback is used, `close(sockfd)` should be called in here for most cases.

Set the user or transport context to NULL if it was freed here, so the server does not try to free it again.

This function is run for all terminated sessions, including sessions where the socket was closed by the network stack - that is, the file descriptor may not be valid anymore.

httpd_uri_match_func_t **uri_match_fn**

URI matcher function.

Called when searching for a matching URI: 1) whose request handler is to be executed right after an HTTP request is successfully parsed 2) in order to prevent duplication while registering a new URI handler using `httpd_register_uri_handler()`

Available options are: 1) NULL : Internally do basic matching using `strncmp()` 2) `httpd_uri_match_wildcard()` : URI wildcard matcher

Users can implement their own matching functions (See description of the `httpd_uri_match_func_t` function prototype)

struct **httpd_req**

HTTP Request Data Structure.

Public Members

httpd_handle_t **handle**

Handle to server instance

int **method**

The type of HTTP request, -1 if unsupported method

const char **uri**[HTTPD_MAX_URI_LEN + 1]

The URI of this request (1 byte extra for null termination)

size_t **content_len**

Length of the request body

void ***aux**

Internally used members

void ***user_ctx**

User context pointer passed during URI registration.

void ***sess_ctx**

Session Context Pointer

A session context. Contexts are maintained across 'sessions' for a given open TCP connection. One session could have multiple request responses. The web server will ensure that the context persists across all these request and responses.

By default, this is NULL. URI Handlers can set this to any meaningful value.

If the underlying socket gets closed, and this pointer is non-NULL, the web server will free up the context by calling free(), unless free_ctx function is set.

httpd_free_ctx_fn_t **free_ctx**

Pointer to free context hook

Function to free session context

If the web server's socket closes, it frees up the session context by calling free() on the sess_ctx member. If you wish to use a custom function for freeing the session context, please specify that here.

bool **ignore_sess_ctx_changes**

Flag indicating if Session Context changes should be ignored

By default, if you change the sess_ctx in some URI handler, the http server will internally free the earlier context (if non NULL), after the URI handler returns. If you want to manage the allocation/reallocation/freeing of sess_ctx yourself, set this flag to true, so that the server will not perform any checks on it. The context will be cleared by the server (by calling free_ctx or free()) only if the socket gets closed.

struct **httpd_uri**

Structure for URI handler.

Public Members

const char ***uri**

The URI to handle

httpd_method_t **method**

Method supported by the URI

esp_err_t (***handler**)(*httpd_req_t* *r)

Handler to call for supported request method. This must return ESP_OK, or else the underlying socket will be closed.

void ***user_ctx**

Pointer to user context data which will be available to handler

Macros

HTTPD_MAX_REQ_HDR_LEN

HTTPD_MAX_URI_LEN

HTTPD_SOCK_ERR_FAIL

HTTPD_SOCK_ERR_INVALID

HTTPD_SOCK_ERR_TIMEOUT

HTTPD_200

HTTP Response 200

HTTPD_204

HTTP Response 204

HTTPD_207

HTTP Response 207

HTTPD_400

HTTP Response 400

HTTPD_404

HTTP Response 404

HTTPD_408

HTTP Response 408

HTTPD_500

HTTP Response 500

HTTPD_TYPE_JSON

HTTP Content type JSON

HTTPD_TYPE_TEXT

HTTP Content type text/HTML

HTTPD_TYPE_OCTET

HTTP Content type octext-stream

ESP_HTTPD_DEF_CTRL_PORT

HTTP Server control socket port

HTTPD_DEFAULT_CONFIG()

ESP_ERR_HTTPD_BASE

Starting number of HTTPD error codes

ESP_ERR_HTTPD_HANDLERS_FULL

All slots for registering URI handlers have been consumed

ESP_ERR_HTTPD_HANDLER_EXISTS

URI handler with same method and target URI already registered

ESP_ERR_HTTPD_INVALID_REQ

Invalid request pointer

ESP_ERR_HTTPD_RESULT_TRUNC

Result string truncated

ESP_ERR_HTTPD_RESP_HDR

Response header field larger than supported

ESP_ERR_HTTPD_RESP_SEND

Error occured while sending response packet

ESP_ERR_HTTPD_ALLOC_MEM

Failed to dynamically allocate memory for resource

ESP_ERR_HTTPD_TASK

Failed to launch server task/thread

HTTPD_RESP_USE_STRLEN

Type Definitions

typedef struct *httpd_req* **httpd_req_t**

HTTP Request Data Structure.

typedef struct *httpd_uri* **httpd_uri_t**

Structure for URI handler.

typedef int (***httpd_send_func_t**)(*httpd_handle_t* hd, int sockfd, const char *buf, size_t buf_len, int flags)

Prototype for HTTPDs low-level send function.

Note: User specified send function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will eventually be conveyed as return value of httpd_send() function

Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Param buf [in] buffer with bytes to send
Param buf_len [in] data size
Param flags [in] flags for the send() function
Return

- Bytes : The number of bytes sent successfully
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket send()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket send()

typedef int (***httpd_recv_func_t**)(*httpd_handle_t* hd, int sockfd, char *buf, size_t buf_len, int flags)

Prototype for HTTPDs low-level recv function.

Note: User specified recv function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will eventually be conveyed as return value of httpd_req_recv() function

Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Param buf [in] buffer with bytes to send
Param buf_len [in] data size
Param flags [in] flags for the send() function
Return

- Bytes : The number of bytes received successfully
- 0 : Buffer length parameter is zero / connection closed by peer
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket recv()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket recv()

typedef int (***httpd_pending_func_t**)(*httpd_handle_t* hd, int sockfd)

Prototype for HTTPDs low-level "get pending bytes" function.

Note: User specified pending function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will be handled accordingly in the server task.

Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Return

- Bytes : The number of bytes waiting to be received
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket pending()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket pending()

typedef *esp_err_t* (***httpd_err_handler_func_t**)(*httpd_req_t* *req, *httpd_err_code_t* error)

Function prototype for HTTP error handling.

This function is executed upon HTTP errors generated during internal processing of an HTTP request. This is used to override the default behavior on error, which is to send HTTP error response and close the underlying socket.

Note:

- If implemented, the server will not automatically send out HTTP error response codes, therefore, httpd_resp_send_err() must be invoked inside this function if user wishes to generate HTTP error responses.
- When invoked, the validity of `uri`, `method`, `content_len` and `user_ctx` fields of the httpd_req_t parameter is not guaranteed as the HTTP request may be partially received/parsed.
- The function must return ESP_OK if underlying socket needs to be kept open. Any other value will ensure that the socket is closed. The return value is ignored when error is of type `HTTPD_500_INTERNAL_SERVER_ERROR` and the socket closed anyway.

Param req [in] HTTP request for which the error needs to be handled
Param error [in] Error type
Return
- ESP_OK : error handled successful
- ESP_FAIL : failure indicates that the underlying socket needs to be closed

typedef void ***httpd_handle_t**

HTTP Server Instance Handle.

Every instance of the server will have a unique handle.

typedef enum http_method **httpd_method_t**

HTTP Method Type wrapper over "enum http_method" available in "http_parser" library.

typedef void (***httpd_free_ctx_fn_t**)(void *ctx)

Prototype for freeing context data (if any)

Param ctx [in] object to free

typedef *esp_err_t* (***httpd_open_func_t**)(*httpd_handle_t* hd, int sockfd)

Function prototype for opening a session.

Called immediately after the socket was opened to set up the send/recv functions and other parameters of the socket.

Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Return
- ESP_OK : On success
- Any value other than ESP_OK will signal the server to close the socket immediately

typedef void (***httpd_close_func_t**)(*httpd_handle_t* hd, int sockfd)

Function prototype for closing a session.

Note: It's possible that the socket descriptor is invalid at this point, the function is called for all terminated sessions. Ensure proper handling of return codes.

Param hd [in] server instance
Param sockfd [in] session socket file descriptor

typedef bool (*`httpd_uri_match_func_t`)(const char *reference_uri, const char *uri_to_match, size_t match_upto)

Function prototype for URI matching.

Param reference_uri [in] URI/template with respect to which the other URI is matched
Param uri_to_match [in] URI/template being matched to the reference URI/template
Param match_upto [in] For specifying the actual length of `uri_to_match` up to which the matching algorithm is to be applied (The maximum value is `strlen(uri_to_match)`, independent of the length of `reference_uri`)
Return true on match

typedef struct *httpd_config* `httpd_config_t`

HTTP Server Configuration Structure.

Note: Use HTTPD_DEFAULT_CONFIG() to initialize the configuration to a default value and then modify only those fields that are specifically determined by the use case.

---

typedef void (*`httpd_work_fn_t`)(void *arg)

Prototype of the HTTPD work function Please refer to httpd_queue_work() for more details.

Param arg [in] The arguments for this work function

Enumerations

enum `httpd_err_code_t`

Error codes sent as HTTP response in case of errors encountered during processing of an HTTP request.

Values:

enumerator `HTTPD_500_INTERNAL_SERVER_ERROR`

enumerator `HTTPD_501_METHOD_NOT_IMPLEMENTED`

enumerator `HTTPD_505_VERSION_NOT_SUPPORTED`

enumerator `HTTPD_400_BAD_REQUEST`

enumerator `HTTPD_401_UNAUTHORIZED`

enumerator `HTTPD_403_FORBIDDEN`

enumerator `HTTPD_404_NOT_FOUND`

enumerator `HTTPD_405_METHOD_NOT_ALLOWED`

enumerator `HTTPD_408_REQ_TIMEOUT`

enumerator `HTTPD_411_LENGTH_REQUIRED`

enumerator **HTTPD_414_URI_TOO_LONG**

enumerator **HTTPD_431_REQ_HDR_FIELDS_TOO_LARGE**

enumerator **HTTPD_ERR_CODE_MAX**

enum **esp_http_server_event_id_t**

HTTP Server events id.

Values:

enumerator **HTTP_SERVER_EVENT_ERROR**

This event occurs when there are any errors during execution

enumerator **HTTP_SERVER_EVENT_START**

This event occurs when HTTP Server is started

enumerator **HTTP_SERVER_EVENT_ON_CONNECTED**

Once the HTTP Server has been connected to the client, no data exchange has been performed

enumerator **HTTP_SERVER_EVENT_ON_HEADER**

Occurs when receiving each header sent from the client

enumerator **HTTP_SERVER_EVENT_HEADERS_SENT**

After sending all the headers to the client

enumerator **HTTP_SERVER_EVENT_ON_DATA**

Occurs when receiving data from the client

enumerator **HTTP_SERVER_EVENT_SENT_DATA**

Occurs when an ESP HTTP server session is finished

enumerator **HTTP_SERVER_EVENT_DISCONNECTED**

The connection has been disconnected

enumerator **HTTP_SERVER_EVENT_STOP**

This event occurs when HTTP Server is stopped

2.2.10 HTTPS Server

Overview

This component is built on top of *HTTP Server*. The HTTPS server takes advantage of hook registration functions in the regular HTTP server to provide callback function for SSL session.

All documentation for *HTTP Server* applies also to a server you create this way.

Used APIs

The following APIs of *HTTP Server* should not be used with *HTTPS Server*, as they are used internally to handle secure sessions and to maintain internal state:

- "send", "receive" and "pending" callback registration functions - secure socket handling
  - `httpd_sess_set_send_override()`
  - `httpd_sess_set_recv_override()`
  - `httpd_sess_set_pending_override()`
- "transport context" - both global and session
  - `httpd_sess_get_transport_ctx()` - returns SSL used for the session
  - `httpd_sess_set_transport_ctx()`
  - `httpd_get_global_transport_ctx()` - returns the shared SSL context
  - `httpd_config::global_transport_ctx`
  - `httpd_config::global_transport_ctx_free_fn`
  - `httpd_config::open_fn` - used to set up secure sockets

Everything else can be used without limitations.

Usage

Please see the example protocols/https_server to learn how to set up a secure server.

Basically, all you need is to generate a certificate, embed it into the firmware, and pass the init struct into the start function after the certificate address and lengths are correctly configured in the init struct.

The server can be started with or without SSL by changing a flag in the init struct - `httpd_ssl_config::transport_mode`. This could be used, e.g., for testing or in trusted environments where you prefer speed over security.

Performance

The initial session setup can take about two seconds, or more with slower clock speed or more verbose logging. Subsequent requests through the open secure socket are much faster (down to under 100 ms).

API Reference

Header File

- components/esp_https_server/include/esp_https_server.h

Functions

esp_err_t **httpd_ssl_start** (*httpd_handle_t* *handle, *httpd_ssl_config_t* *config)

Create a SSL capable HTTP server (secure mode may be disabled in config)

Parameters

- **config** –**[inout]** - server config, must not be const. Does not have to stay valid after calling this function.
- **handle** –**[out]** - storage for the server handle, must be a valid pointer

Returns success

esp_err_t **httpd_ssl_stop** (*httpd_handle_t* handle)

Stop the server. Blocks until the server is shut down.

Parameters **handle** –**[in]**

Returns

- ESP_OK: Server stopped successfully
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_FAIL: Failure to shut down server

Structures

struct **esp_https_server_user_cb_arg**

Callback data struct, contains the ESP-TLS connection handle and the connection state at which the callback is executed.

Public Members

httpd_ssl_user_cb_state_t **user_cb_state**

State of user callback

esp_tls_t ***tls**

ESP-TLS connection handle

struct **httpd_ssl_config**

HTTPS server config struct

Please use HTTPD_SSL_CONFIG_DEFAULT() to initialize it.

Public Members

httpd_config_t **httpd**

Underlying HTTPD server config

Parameters like task stack size and priority can be adjusted here.

const uint8_t ***servercert**

Server certificate

size_t **servercert_len**

Server certificate byte length

const uint8_t ***cacert_pem**

CA certificate ((CA used to sign clients, or client cert itself)

size_t **cacert_len**

CA certificate byte length

const uint8_t ***prvtkey_pem**

Private key

size_t **prvtkey_len**

Private key byte length

httpd_ssl_transport_mode_t **transport_mode**

Transport Mode (default secure)

uint16_t **port_secure**

Port used when transport mode is secure (default 443)

uint16_t **port_insecure**

Port used when transport mode is insecure (default 80)

bool **session_tickets**

Enable tls session tickets

bool **use_secure_element**

Enable secure element for server session

esp_https_server_user_cb ***user_cb**

User callback for esp_https_server

void ***ssl_userdata**

user data to add to the ssl context

esp_tls_handshake_callback **cert_select_cb**

Certificate selection callback to use

Macros

HTTPD_SSL_CONFIG_DEFAULT()

Default config struct init

(http_server default config had to be copied for customization)

Notes:

- port is set when starting the server, according to 'transport_mode'
- one socket uses ~ 40kB RAM with SSL, we reduce the default socket count to 4
- SSL sockets are usually long-lived, closing LRU prevents pool exhaustion DOS
- Stack size may need adjustments depending on the user application

Type Definitions

typedef struct *esp_https_server_user_cb_arg* **esp_https_server_user_cb_arg_t**

Callback data struct, contains the ESP-TLS connection handle and the connection state at which the callback is executed.

typedef void **esp_https_server_user_cb**(*esp_https_server_user_cb_arg_t* *user_cb)

Callback function prototype Can be used to get connection or client information (SSL context) E.g. Client certificate, Socket FD, Connection state, etc.

Param user_cb Callback data struct

typedef struct *httpd_ssl_config* **httpd_ssl_config_t**

Enumerations

enum **httpd_ssl_transport_mode_t**

Values:

enumerator **HTTPD_SSL_TRANSPORT_SECURE**

enumerator **HTTPD_SSL_TRANSPORT_INSECURE**

enum **httpd_ssl_user_cb_state_t**

Indicates the state at which the user callback is executed, i.e at session creation or session close.

Values:

enumerator **HTTPD_SSL_USER_CB_SESS_CREATE**

enumerator **HTTPD_SSL_USER_CB_SESS_CLOSE**

2.2.11 ICMP Echo

Overview

ICMP (Internet Control Message Protocol) is used for diagnostic or control purposes or generated in response to errors in IP operations. The common network util `ping` is implemented based on the ICMP packets with the type field value of 0, also called `Echo Reply`.

During a ping session, the source host firstly sends out an ICMP echo request packet and wait for an ICMP echo reply with specific times. In this way, it also measures the round-trip time for the messages. After receiving a valid ICMP echo reply, the source host will generate statistics about the IP link layer (e.g. packet loss, elapsed time, etc).

It is common that IoT device needs to check whether a remote server is alive or not. The device should show the warnings to users when it got offline. It can be achieved by creating a ping session and sending/parsing ICMP echo packets periodically.

To make this internal procedure much easier for users, ESP-IDF provides some out-of-box APIs.

Create a new ping session To create a ping session, you need to fill in the `esp_ping_config_t` configuration structure firstly, specifying target IP address, interval times, and etc. Optionally, you can also register some callback functions with the *esp_ping_callbacks_t*` structure.

Example method to create a new ping session and register callbacks:

```
static void test_on_ping_success(esp_ping_handle_t hdl, void *args)
{
    // optionally, get callback arguments
    // const char* str = (const char*) args;
    // printf("%s\r\n", str); // "foo"
    uint8_t ttl;
    uint16_t seqno;
    uint32_t elapsed_time, recv_len;
    ip_addr_t target_addr;
    esp_ping_get_profile(hdl, ESP_PING_PROF_SEQNO, &seqno, sizeof(seqno));
    esp_ping_get_profile(hdl, ESP_PING_PROF_TTL, &ttl, sizeof(ttl));
    esp_ping_get_profile(hdl, ESP_PING_PROF_IPADDR, &target_addr, sizeof(target_
↪addr));
    esp_ping_get_profile(hdl, ESP_PING_PROF_SIZE, &recv_len, sizeof(recv_len));
    esp_ping_get_profile(hdl, ESP_PING_PROF_TIMEGAP, &elapsed_time, sizeof(elapsed_
↪time));
    printf("%d bytes from %s icmp_seq=%d ttl=%d time=%d ms\n",
           recv_len, inet_ntoa(target_addr.u_addr.ip4), seqno, ttl, elapsed_time);
}

static void test_on_ping_timeout(esp_ping_handle_t hdl, void *args)
{
    uint16_t seqno;
    ip_addr_t target_addr;
    esp_ping_get_profile(hdl, ESP_PING_PROF_SEQNO, &seqno, sizeof(seqno));
```

(continues on next page)

(continued from previous page)

```
    esp_ping_get_profile(hdl, ESP_PING_PROF_IPADDR, &target_addr, sizeof(target_
↪addr));
    printf("From %s icmp_seq=%d timeout\n", inet_ntoa(target_addr.u_addr.ip4),␣
↪seqno);
}

static void test_on_ping_end(esp_ping_handle_t hdl, void *args)
{
    uint32_t transmitted;
    uint32_t received;
    uint32_t total_time_ms;

    esp_ping_get_profile(hdl, ESP_PING_PROF_REQUEST, &transmitted,␣
↪sizeof(transmitted));
    esp_ping_get_profile(hdl, ESP_PING_PROF_REPLY, &received, sizeof(received));
    esp_ping_get_profile(hdl, ESP_PING_PROF_DURATION, &total_time_ms, sizeof(total_
↪time_ms));
    printf("%d packets transmitted, %d received, time %dms\n", transmitted,␣
↪received, total_time_ms);
}

void initialize_ping()
{
    /* convert URL to IP address */
    ip_addr_t target_addr;
    struct addrinfo hint;
    struct addrinfo *res = NULL;
    memset(&hint, 0, sizeof(hint));
    memset(&target_addr, 0, sizeof(target_addr));
    getaddrinfo("www.espressif.com", NULL, &hint, &res);
    struct in_addr addr4 = ((struct sockaddr_in *) (res->ai_addr))->sin_addr;
    inet_addr_to_ip4addr(ip_2_ip4(&target_addr), &addr4);
    freeaddrinfo(res);

    esp_ping_config_t ping_config = ESP_PING_DEFAULT_CONFIG();
    ping_config.target_addr = target_addr;          // target IP address
    ping_config.count = ESP_PING_COUNT_INFINITE;    // ping in infinite mode, esp_
↪ping_stop can stop it

    /* set callback functions */
    esp_ping_callbacks_t cbs;
    cbs.on_ping_success = test_on_ping_success;
    cbs.on_ping_timeout = test_on_ping_timeout;
    cbs.on_ping_end = test_on_ping_end;
    cbs.cb_args = "foo";  // arguments that will feed to all callback functions,␣
↪can be NULL
    cbs.cb_args = eth_event_group;

    esp_ping_handle_t ping;
    esp_ping_new_session(&ping_config, &cbs, &ping);
}
```

Start and Stop ping session You can start and stop ping session with the handle returned by `esp_ping_new_session`. Note that, the ping session won't start automatically after creation. If the ping session is stopped, and restart again, the sequence number in ICMP packets will recount from zero again.

Delete a ping session If a ping session won't be used any more, you can delete it with `esp_ping_delete_session`. Please make sure the ping session is in stop state (i.e. you have called `esp_ping_stop` before or the ping session has finished all the procedures) when you call this function.

Get runtime statistics As the example code above, you can call `esp_ping_get_profile` to get different runtime statistics of ping session in the callback function.

Application Example

ICMP echo example: protocols/icmp_echo

API Reference

Header File

- components/lwip/include/apps/ping/ping_sock.h

Functions

esp_err_t **esp_ping_new_session**(const *esp_ping_config_t* *config, const *esp_ping_callbacks_t* *cbs, *esp_ping_handle_t* *hdl_out)

Create a ping session.

Parameters
- **config** –ping configuration
- **cbs** –a bunch of callback functions invoked by internal ping task
- **hdl_out** –handle of ping session

Returns
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. configuration is null, etc)
- ESP_ERR_NO_MEM: out of memory
- ESP_FAIL: other internal error (e.g. socket error)
- ESP_OK: create ping session successfully, user can take the ping handle to do follow-on jobs

esp_err_t **esp_ping_delete_session**(*esp_ping_handle_t* hdl)

Delete a ping session.

Parameters **hdl** –handle of ping session

Returns
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: delete ping session successfully

esp_err_t **esp_ping_start**(*esp_ping_handle_t* hdl)

Start the ping session.

Parameters **hdl** –handle of ping session

Returns
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: start ping session successfully

esp_err_t **esp_ping_stop**(*esp_ping_handle_t* hdl)

Stop the ping session.

Parameters **hdl** –handle of ping session

Returns
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: stop ping session successfully

esp_err_t **esp_ping_get_profile**(*esp_ping_handle_t* hdl, *esp_ping_profile_t* profile, void *data, uint32_t size)

Get runtime profile of ping session.

Parameters
- **hdl** –handle of ping session
- **profile** –type of profile

- **data** –profile data
- **size** –profile data size

Returns

- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_ERR_INVALID_SIZE: the actual profile data size doesn't match the "size" parameter
- ESP_OK: get profile successfully

Structures

struct **esp_ping_callbacks_t**

Type of "ping" callback functions.

Public Members

void ***cb_args**

arguments for callback functions

void (***on_ping_success**)(*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when received ICMP echo reply packet.

void (***on_ping_timeout**)(*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when receive ICMP echo reply packet timeout.

void (***on_ping_end**)(*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when a ping session is finished.

struct **esp_ping_config_t**

Type of "ping" configuration.

Public Members

uint32_t **count**

A "ping" session contains count procedures

uint32_t **interval_ms**

Milliseconds between each ping procedure

uint32_t **timeout_ms**

Timeout value (in milliseconds) of each ping procedure

uint32_t **data_size**

Size of the data next to ICMP packet header

int **tos**

Type of Service, a field specified in the IP header

int **ttl**

Time to Live,a field specified in the IP header

ip_addr_t **target_addr**

Target IP address, either IPv4 or IPv6

uint32_t **task_stack_size**

Stack size of internal ping task

uint32_t **task_prio**

Priority of internal ping task

uint32_t **interface**

Netif index, interface=0 means NETIF_NO_INDEX

Macros

ESP_PING_DEFAULT_CONFIG()

Default ping configuration.

ESP_PING_COUNT_INFINITE

Set ping count to zero will ping target infinitely

Type Definitions

typedef void ***esp_ping_handle_t**

Type of "ping" session handle.

Enumerations

enum **esp_ping_profile_t**

Profile of ping session.

Values:

enumerator **ESP_PING_PROF_SEQNO**

Sequence number of a ping procedure

enumerator **ESP_PING_PROF_TOS**

Type of service of a ping procedure

enumerator **ESP_PING_PROF_TTL**

Time to live of a ping procedure

enumerator **ESP_PING_PROF_REQUEST**

Number of request packets sent out

enumerator **ESP_PING_PROF_REPLY**

Number of reply packets received

enumerator **ESP_PING_PROF_IPADDR**

IP address of replied target

enumerator **ESP_PING_PROF_SIZE**

Size of received packet

enumerator **ESP_PING_PROF_TIMEGAP**

Elapsed time between request and reply packet

enumerator **ESP_PING_PROF_DURATION**

Elapsed time of the whole ping session

2.2.12 mDNS Service

mDNS is a multicast UDP service that is used to provide local network service and host discovery.

The ESP-IDF component *mDNS* has been moved from ESP-IDF since version v5.0 to a separate repository:

- mDNS component on GitHub

To add mDNS component in your project, please run *idf.py add-dependency espressif/mdns*.

Hosted Documentation

The documentation can be found on the link below:

- mDNS documentation

2.2.13 Mbed TLS

Mbed TLS is a C library that implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. Its small code footprint makes it suitable for embedded systems.

Note: ESP-IDF uses a fork of Mbed TLS which includes a few patches (related to hardware routines of certain modules like `bignum (MPI)` and `ECC`) over vanilla Mbed TLS.

Mbed TLS supports SSL 3.0 up to TLS 1.3 and DTLS 1.0 to 1.2 communication by providing the following:

- TCP/IP communication functions: listen, connect, accept, read/write.
- SSL/TLS communication functions: init, handshake, read/write.
- X.509 functions: CRT, CRL and key handling
- Random number generation
- Hashing
- Encryption/decryption

Note: Mbed TLS is in the process of migrating all the documentation to a single place. In the meantime, users can find the documentation at the old Mbed TLS site .

Mbed TLS Support in ESP-IDF

Please find the information about the Mbed TLS versions present in different branches of ESP-IDF here.

Note: Please refer the *ESP-IDF Migration Guide* to migrate from Mbed TLS version 2.x to version 3.0 or greater.

Application Examples

Examples in ESP-IDF use *ESP-TLS* which provides a simplified API interface for accessing the commonly used TLS functionality.

Refer to the examples protocols/https_server/simple (Simple HTTPS server) and protocols/https_request (Make HTTPS requests) for more information.

If the Mbed TLS API is to be used directly, refer to the example protocols/https_mbedtls.

Alternatives

ESP-TLS acts as an abstraction layer over the underlying SSL/TLS library and thus has an option to use Mbed TLS or wolfSSL as the underlying library. By default, only Mbed TLS is available and used in ESP-IDF whereas wolfSSL is available publicly at https://github.com/espressif/esp-wolfSSL with the upstream submodule pointer.

Please refer to *ESP-TLS: Underlying SSL/TLS Library Options* docs for more information on this and comparison of Mbed TLS and wolfSSL.

Important Config Options

Following is a brief list of important config options accessible at `Component Config -> mbedTLS`. The full list of config options can be found *here*.

- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_2*: Support for TLS 1.2
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*: Support for TLS 1.3
- *CONFIG_MBEDTLS_CERTIFICATE_BUNDLE*: Support for trusted root certificate bundle (more about this: *ESP x509 Certificate Bundle*)
- *CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS*: Support for TLS Session Resumption: Client session tickets
- *CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS*: Support for TLS Session Resumption: Server session tickets
- *CONFIG_MBEDTLS_HARDWARE_SHA*: Support for hardware SHA acceleration
- *CONFIG_MBEDTLS_HARDWARE_AES*: Support for hardware AES acceleration
- *CONFIG_MBEDTLS_HARDWARE_MPI*: Support for hardware MPI (bignum) acceleration

---

Note: Mbed TLS v3.0.0 and later support only TLS 1.2 and TLS 1.3 (SSL 3.0, TLS 1.0, TLS 1.1 and DTLS 1.0 are not supported). The support for TLS 1.3 is experimental and only supports the client-side. More information about this can be found out here.

Performance and Memory Tweaks

Reducing Heap Usage The following table shows typical memory usage with different configs when the protocols/https_request example (with Server Validation enabled) was run with Mbed TLS as the SSL/TLS library.

Mbed TLS Test	Related Configs	Heap Usage (approx.)
Default	NA	42196 B
Enable SSL Variable Length	*CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH*	42120 B
Disable Keep Peer Certificate	*CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE*	38533 B
Enable Dynamic TX/RX Buffer	*CONFIG_MBEDTLS_DYNAMIC_BUFFER* *CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA* *CONFIG_MBEDTLS_DYNAMIC_FREE_CA_CERT*	22013 B

Note: These values are subject to change with change in configuration options and versions of Mbed TLS.

Reducing Binary Size Under `Component Config -> mbedTLS`, there are multiple Mbed TLS features which are enabled by default but can be disabled if not needed to save code size. More information can be about this can be found in *Minimizing Binary Size* docs.

Code examples for this API section are provided in the protocols directory of ESP-IDF examples.

2.2.14 IP Network Layer

Documentation for IP Network Layer protocols (below the Application Protocol layer) are provided in *Networking APIs*.

2.3 Bluetooth API

2.3.1 BT COMMON

BT GENERIC DEFINES

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_bt_defs.h

Structures

struct **esp_bt_uuid_t**

UUID type.

Public Members

uint16_t **len**

UUID length, 16bit, 32bit or 128bit

uint16_t **uuid16**
: 16bit UUID

uint32_t **uuid32**
: 32bit UUID

uint8_t **uuid128**[ESP_UUID_LEN_128]
: 128bit UUID

union *esp_bt_uuid_t*::[anonymous] **uuid**
: UUID

Macros

ESP_BLUEDROID_STATUS_CHECK(status)

ESP_BT_STATUS_BASE_FOR_HCI_ERR

ESP_BT_OCTET16_LEN

ESP_BT_OCTET8_LEN

ESP_DEFAULT_GATT_IF
: Default GATT interface id.

ESP_BLE_PRIM_ADV_INT_MIN
: Minimum advertising interval for undirected and low duty cycle directed advertising

ESP_BLE_PRIM_ADV_INT_MAX
: Maximum advertising interval for undirected and low duty cycle directed advertising

ESP_BLE_CONN_INT_MIN
: relate to BTM_BLE_CONN_INT_MIN in stack/btm_ble_api.h

ESP_BLE_CONN_INT_MAX
: relate to BTM_BLE_CONN_INT_MAX in stack/btm_ble_api.h

ESP_BLE_CONN_LATENCY_MAX
: relate to ESP_BLE_CONN_LATENCY_MAX in stack/btm_ble_api.h

ESP_BLE_CONN_SUP_TOUT_MIN
: relate to BTM_BLE_CONN_SUP_TOUT_MIN in stack/btm_ble_api.h

ESP_BLE_CONN_SUP_TOUT_MAX
: relate to ESP_BLE_CONN_SUP_TOUT_MAX in stack/btm_ble_api.h

ESP_BLE_IS_VALID_PARAM(x, min, max)
: Check the param is valid or not.

ESP_UUID_LEN_16

ESP_UUID_LEN_32

ESP_UUID_LEN_128

ESP_BD_ADDR_LEN

Bluetooth address length.

ESP_BLE_ENC_KEY_MASK

Used to exchange the encryption key in the init key & response key.

ESP_BLE_ID_KEY_MASK

Used to exchange the IRK key in the init key & response key.

ESP_BLE_CSR_KEY_MASK

Used to exchange the CSRK key in the init key & response key.

ESP_BLE_LINK_KEY_MASK

Used to exchange the link key(this key just used in the BLE & BR/EDR coexist mode) in the init key & response key.

ESP_APP_ID_MIN

Minimum of the application id.

ESP_APP_ID_MAX

Maximum of the application id.

ESP_BD_ADDR_STR

ESP_BD_ADDR_HEX(addr)

Type Definitions

typedef uint8_t **esp_bt_octet16_t**[ESP_BT_OCTET16_LEN]

typedef uint8_t **esp_bt_octet8_t**[ESP_BT_OCTET8_LEN]

typedef uint8_t **esp_link_key**[ESP_BT_OCTET16_LEN]

typedef uint8_t **esp_bd_addr_t**[ESP_BD_ADDR_LEN]

Bluetooth device address.

typedef uint8_t **esp_ble_key_mask_t**

Enumerations

enum **esp_bt_status_t**

Status Return Value.

Values:

enumerator **ESP_BT_STATUS_SUCCESS**

enumerator **ESP_BT_STATUS_FAIL**

enumerator **ESP_BT_STATUS_NOT_READY**

enumerator **ESP_BT_STATUS_NOMEM**

enumerator **ESP_BT_STATUS_BUSY**

enumerator **ESP_BT_STATUS_DONE**

enumerator **ESP_BT_STATUS_UNSUPPORTED**

enumerator **ESP_BT_STATUS_PARM_INVALID**

enumerator **ESP_BT_STATUS_UNHANDLED**

enumerator **ESP_BT_STATUS_AUTH_FAILURE**

enumerator **ESP_BT_STATUS_RMT_DEV_DOWN**

enumerator **ESP_BT_STATUS_AUTH_REJECTED**

enumerator **ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR**

enumerator **ESP_BT_STATUS_PENDING**

enumerator **ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL**

enumerator **ESP_BT_STATUS_PARAM_OUT_OF_RANGE**

enumerator **ESP_BT_STATUS_TIMEOUT**

enumerator **ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED**

enumerator **ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED**

enumerator **ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT**

enumerator **ESP_BT_STATUS_MEMORY_FULL**

enumerator **ESP_BT_STATUS_EIR_TOO_LARGE**

enumerator **ESP_BT_STATUS_HCI_SUCCESS**

enumerator **ESP_BT_STATUS_HCI_ILLEGAL_COMMAND**

enumerator **ESP_BT_STATUS_HCI_NO_CONNECTION**

enumerator **ESP_BT_STATUS_HCI_HW_FAILURE**

enumerator **ESP_BT_STATUS_HCI_PAGE_TIMEOUT**

enumerator **ESP_BT_STATUS_HCI_AUTH_FAILURE**

enumerator **ESP_BT_STATUS_HCI_KEY_MISSING**

enumerator **ESP_BT_STATUS_HCI_MEMORY_FULL**

enumerator **ESP_BT_STATUS_HCI_CONNECTION_TOUT**

enumerator **ESP_BT_STATUS_HCI_MAX_NUM_OF_CONNECTIONS**

enumerator **ESP_BT_STATUS_HCI_MAX_NUM_OF_SCOS**

enumerator **ESP_BT_STATUS_HCI_CONNECTION_EXISTS**

enumerator **ESP_BT_STATUS_HCI_COMMAND_DISALLOWED**

enumerator **ESP_BT_STATUS_HCI_HOST_REJECT_RESOURCES**

enumerator **ESP_BT_STATUS_HCI_HOST_REJECT_SECURITY**

enumerator **ESP_BT_STATUS_HCI_HOST_REJECT_DEVICE**

enumerator **ESP_BT_STATUS_HCI_HOST_TIMEOUT**

enumerator **ESP_BT_STATUS_HCI_UNSUPPORTED_VALUE**

enumerator **ESP_BT_STATUS_HCI_ILLEGAL_PARAMETER_FMT**

enumerator **ESP_BT_STATUS_HCI_PEER_USER**

enumerator **ESP_BT_STATUS_HCI_PEER_LOW_RESOURCES**

enumerator **ESP_BT_STATUS_HCI_PEER_POWER_OFF**

enumerator **ESP_BT_STATUS_HCI_CONN_CAUSE_LOCAL_HOST**

enumerator **ESP_BT_STATUS_HCI_REPEATED_ATTEMPTS**

enumerator **ESP_BT_STATUS_HCI_PAIRING_NOT_ALLOWED**

enumerator **ESP_BT_STATUS_HCI_UNKNOWN_LMP_PDU**

enumerator **ESP_BT_STATUS_HCI_UNSUPPORTED_REM_FEATURE**

enumerator **ESP_BT_STATUS_HCI_SCO_OFFSET_REJECTED**

enumerator **ESP_BT_STATUS_HCI_SCO_INTERVAL_REJECTED**

enumerator **ESP_BT_STATUS_HCI_SCO_AIR_MODE**

enumerator **ESP_BT_STATUS_HCI_INVALID_LMP_PARAM**

enumerator **ESP_BT_STATUS_HCI_UNSPECIFIED**

enumerator **ESP_BT_STATUS_HCI_UNSUPPORTED_LMP_PARAMETERS**

enumerator **ESP_BT_STATUS_HCI_ROLE_CHANGE_NOT_ALLOWED**

enumerator **ESP_BT_STATUS_HCI_LMP_RESPONSE_TIMEOUT**

enumerator **ESP_BT_STATUS_HCI_LMP_ERR_TRANS_COLLISION**

enumerator **ESP_BT_STATUS_HCI_LMP_PDU_NOT_ALLOWED**

enumerator **ESP_BT_STATUS_HCI_ENCRY_MODE_NOT_ACCEPTABLE**

enumerator **ESP_BT_STATUS_HCI_UNIT_KEY_USED**

enumerator **ESP_BT_STATUS_HCI_QOS_NOT_SUPPORTED**

enumerator **ESP_BT_STATUS_HCI_INSTANT_PASSED**

enumerator **ESP_BT_STATUS_HCI_PAIRING_WITH_UNIT_KEY_NOT_SUPPORTED**

enumerator **ESP_BT_STATUS_HCI_DIFF_TRANSACTION_COLLISION**

enumerator **ESP_BT_STATUS_HCI_UNDEFINED_0x2B**

enumerator **ESP_BT_STATUS_HCI_QOS_UNACCEPTABLE_PARAM**

enumerator **ESP_BT_STATUS_HCI_QOS_REJECTED**

enumerator **ESP_BT_STATUS_HCI_CHAN_CLASSIF_NOT_SUPPORTED**

enumerator **ESP_BT_STATUS_HCI_INSUFFCIENT_SECURITY**

enumerator **ESP_BT_STATUS_HCI_PARAM_OUT_OF_RANGE**

enumerator **ESP_BT_STATUS_HCI_UNDEFINED_0x31**

enumerator **ESP_BT_STATUS_HCI_ROLE_SWITCH_PENDING**

enumerator **ESP_BT_STATUS_HCI_UNDEFINED_0x33**

enumerator **ESP_BT_STATUS_HCI_RESERVED_SLOT_VIOLATION**

enumerator **ESP_BT_STATUS_HCI_ROLE_SWITCH_FAILED**

enumerator **ESP_BT_STATUS_HCI_INQ_RSP_DATA_TOO_LARGE**

enumerator **ESP_BT_STATUS_HCI_SIMPLE_PAIRING_NOT_SUPPORTED**

enumerator **ESP_BT_STATUS_HCI_HOST_BUSY_PAIRING**

enumerator **ESP_BT_STATUS_HCI_REJ_NO_SUITABLE_CHANNEL**

enumerator **ESP_BT_STATUS_HCI_CONTROLLER_BUSY**

enumerator **ESP_BT_STATUS_HCI_UNACCEPT_CONN_INTERVAL**

enumerator **ESP_BT_STATUS_HCI_DIRECTED_ADVERTISING_TIMEOUT**

enumerator **ESP_BT_STATUS_HCI_CONN_TOUT_DUE_TO_MIC_FAILURE**

enumerator **ESP_BT_STATUS_HCI_CONN_FAILED_ESTABLISHMENT**

enumerator **ESP_BT_STATUS_HCI_MAC_CONNECTION_FAILED**

enum **esp_bt_dev_type_t**

Bluetooth device type.

Values:

enumerator **ESP_BT_DEVICE_TYPE_BREDR**

enumerator **ESP_BT_DEVICE_TYPE_BLE**

enumerator **ESP_BT_DEVICE_TYPE_DUMO**

enum **esp_ble_addr_type_t**

BLE device address type.

Values:

enumerator **BLE_ADDR_TYPE_PUBLIC**

Public Device Address

enumerator **BLE_ADDR_TYPE_RANDOM**

Random Device Address. To set this address, use the function esp_ble_gap_set_rand_addr(esp_bd_addr_t rand_addr)

enumerator **BLE_ADDR_TYPE_RPA_PUBLIC**

Resolvable Private Address (RPA) with public identity address

enumerator **BLE_ADDR_TYPE_RPA_RANDOM**

Resolvable Private Address (RPA) with random identity address. To set this address, use the function esp_ble_gap_set_rand_addr(esp_bd_addr_t rand_addr)

enum **esp_ble_wl_addr_type_t**

white list address type

Values:

enumerator **BLE_WL_ADDR_TYPE_PUBLIC**

enumerator **BLE_WL_ADDR_TYPE_RANDOM**

BT MAIN API

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_bt_main.h

Functions

esp_bluedroid_status_t **esp_bluedroid_get_status**(void)

Get bluetooth stack status.

Returns Bluetooth stack status

esp_err_t **esp_bluedroid_enable**(void)

Enable bluetooth, must after esp_bluedroid_init().

Returns

- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_disable**(void)

Disable bluetooth, must prior to esp_bluedroid_deinit().

Returns

- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_init**(void)

Init and alloc the resource for bluetooth, must be prior to every bluetooth stuff.

Returns
- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_deinit**(void)

Deinit and free the resource for bluetooth, must be after every bluetooth stuff.

Returns
- ESP_OK : Succeed
- Other : Failed

Enumerations

enum **esp_bluedroid_status_t**

Bluetooth stack status type, to indicate whether the bluetooth stack is ready.

Values:

enumerator **ESP_BLUEDROID_STATUS_UNINITIALIZED**

Bluetooth not initialized

enumerator **ESP_BLUEDROID_STATUS_INITIALIZED**

Bluetooth initialized but not enabled

enumerator **ESP_BLUEDROID_STATUS_ENABLED**

Bluetooth initialized and enabled

BT DEVICE APIs

Overview Bluetooth device reference APIs.

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_bt_device.h

Functions

const uint8_t ***esp_bt_dev_get_address**(void)

Get bluetooth device address. Must use after "esp_bluedroid_enable" .

Returns bluetooth device address (six bytes), or NULL if bluetooth stack is not enabled

esp_err_t **esp_bt_dev_set_device_name**(const char *name)

Set bluetooth device name. This function should be called after esp_bluedroid_enable() completes successfully.

A BR/EDR/LE device type shall have a single Bluetooth device name which shall be identical irrespective of the physical channel used to perform the name discovery procedure.

Parameters **name** –**[in]** : device name to be set

Returns
- ESP_OK : Succeed
- ESP_ERR_INVALID_ARG : if name is NULL pointer or empty, or string length out of limit

- ESP_ERR_INVALID_STATE : if bluetooth stack is not yet enabled
- ESP_FAIL : others

2.3.2 BT LE

GAP API

Application Example Check bluetooth/bluedroid/ble folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a SMP security client demo and its tutorial. This demo initiates its security parameters and acts as a GATT client, which can send a security request to the peer device and then complete the encryption procedure.
  - bluetooth/bluedroid/ble/gatt_security_client
  - GATT Security Client Example Walkthrough
- This is a SMP security server demo and its tutorial. This demo initiates its security parameters and acts as a GATT server, which can send a pair request to the peer device and then complete the encryption procedure.
  - bluetooth/bluedroid/ble/gatt_security_server
  - GATT Security Server Example Walkthrough

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_gap_ble_api.h

Functions

esp_err_t **esp_ble_gap_register_callback**(*esp_gap_ble_cb_t* callback)

This function is called to occur gap event, such as scan result.

Parameters **callback** –**[in]** callback function

Returns

- ESP_OK : success
- other : failed

esp_gap_ble_cb_t **esp_ble_gap_get_callback**(void)

This function is called to get the current gap callback.

Returns

- esp_gap_ble_cb_t : callback function

esp_err_t **esp_ble_gap_config_adv_data**(*esp_ble_adv_data_t* *adv_data)

This function is called to override the BTA default ADV parameters.

Parameters **adv_data** –**[in]** Pointer to User defined ADV data structure. This memory space can not be freed until callback of config_adv_data is received.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_scan_params**(*esp_ble_scan_params_t* *scan_params)

This function is called to set scan parameters.

Parameters **scan_params** –**[in]** Pointer to User defined scan_params data structure. This memory space can not be freed until callback of set_scan_params

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_start_scanning**(uint32_t duration)

This procedure keep the device scanning the peer device which advertising on the air.

Parameters **duration** –**[in]** Keeping the scanning time, the unit is second.
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_stop_scanning**(void)

This function call to stop the device scanning the peer device which advertising on the air.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_start_advertising**(*esp_ble_adv_params_t* *adv_params)

This function is called to start advertising.

Parameters **adv_params** –**[in]** pointer to User defined adv_params data structure.
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_stop_advertising**(void)

This function is called to stop advertising.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_update_conn_params**(*esp_ble_conn_update_params_t* *params)

Update connection parameters, can only be used when connection is up.

Parameters **params** –**[in]** - connection update parameters
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_pkt_data_len**(*esp_bd_addr_t* remote_device, uint16_t tx_data_length)

This function is to set maximum LE data packet size.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_rand_addr**(*esp_bd_addr_t* rand_addr)

This function allows configuring either a Non-Resolvable Private Address or a Static Random Address.

Parameters **rand_addr** –**[in]** The address to be configured. Refer to the table below for possible address subtypes:

address [47:46]	Address Type
0b00	Non-Resolvable Private Address
0b11	Static Random Address

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_clear_rand_addr**(void)

This function clears the random address for the application.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_local_privacy**(bool privacy_enable)

Enable/disable privacy (including address resolution) on the local device.

Parameters **privacy_enable** –**[in]** - enable/disable privacy on remote device.
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_local_icon**(uint16_t icon)

set local gap appearance icon

Parameters **icon** –**[in]** - External appearance value, these values are defined by the Bluetooth SIG, please refer to https://www.bluetooth.com/specifications/assigned-numbers/
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_update_whitelist**(bool add_remove, *esp_bd_addr_t* remote_bda, *esp_ble_wl_addr_type_t* wl_addr_type)

Add or remove device from white list.

Parameters
- **add_remove** –**[in]** the value is true if added the ble device to the white list, and false remove to the white list.
- **remote_bda** –**[in]** the remote device address add/remove from the white list.
- **wl_addr_type** –**[in]** whitelist address type

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_clear_whitelist**(void)

Clear all white list.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_get_whitelist_size**(uint16_t *length)

Get the whitelist size in the controller.

Parameters **length** –**[out]** the white list length.
Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_prefer_conn_params**(*esp_bd_addr_t* bd_addr, uint16_t min_conn_int, uint16_t max_conn_int, uint16_t slave_latency, uint16_t supervision_tout)

This function is called to set the preferred connection parameters when default connection parameter is not desired before connecting. This API can only be used in the master role.

Parameters
- **bd_addr** –**[in]** BD address of the peripheral
- **min_conn_int** –**[in]** minimum preferred connection interval
- **max_conn_int** –**[in]** maximum preferred connection interval
- **slave_latency** –**[in]** preferred slave latency

- **supervision_tout** –**[in]** preferred supervision timeout

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_device_name**(const char *name)

Set device name to the local device Note: This API don' t affect the advertising data.

Parameters **name** –**[in]** - device name.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_get_device_name**(void)

Get device name of the local device.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_get_local_used_addr**(*esp_bd_addr_t* local_used_addr, uint8_t *addr_type)

This function is called to get local used address and address type. uint8_t *esp_bt_dev_get_address(void) get the public address.

Parameters

- **local_used_addr** –**[in]** - current local used ble address (six bytes)
- **addr_type** –**[in]** - ble address type

Returns - ESP_OK : success

- other : failed

uint8_t ***esp_ble_resolve_adv_data**(uint8_t *adv_data, uint8_t type, uint8_t *length)

This function is called to get ADV data for a specific type.

Parameters

- **adv_data** –**[in]** - pointer of ADV data which to be resolved
- **type** –**[in]** - finding ADV data type
- **length** –**[out]** - return the length of ADV data not including type

Returns pointer of ADV data

esp_err_t **esp_ble_gap_config_adv_data_raw**(uint8_t *raw_data, uint32_t raw_data_len)

This function is called to set raw advertising data. User need to fill ADV data by self.

Parameters

- **raw_data** –**[in]** : raw advertising data with the format: [Length 1][Data Type 1][Data 1][Length 2][Data Type 2][Data 2] ···
- **raw_data_len** –**[in]** : raw advertising data length , less than 31 bytes

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_scan_rsp_data_raw**(uint8_t *raw_data, uint32_t raw_data_len)

This function is called to set raw scan response data. User need to fill scan response data by self.

Parameters

- **raw_data** –**[in]** : raw scan response data
- **raw_data_len** –**[in]** : raw scan response data length , less than 31 bytes

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_read_rssi**(*esp_bd_addr_t* remote_addr)

This function is called to read the RSSI of remote device. The address of link policy results are returned in the gap callback function with ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT event.

Parameters **remote_addr** –**[in]** : The remote connection device address.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_add_duplicate_scan_exceptional_device**(*esp_ble_duplicate_exceptional_info_type_t* type, *esp_duplicate_info_t* device_info)

This function is called to add a device info into the duplicate scan exceptional list.

Parameters

- **type** –**[in]** device info type, it is defined in esp_ble_duplicate_exceptional_info_type_t when type is MESH_BEACON_TYPE, MESH_PROV_SRV_ADV or MESH_PROXY_SRV_ADV , device_info is invalid.
- **device_info** –**[in]** the device information.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_remove_duplicate_scan_exceptional_device**(*esp_ble_duplicate_exceptional_info_type_t* type, *esp_duplicate_info_t* device_info)

This function is called to remove a device info from the duplicate scan exceptional list.

Parameters

- **type** –**[in]** device info type, it is defined in esp_ble_duplicate_exceptional_info_type_t when type is MESH_BEACON_TYPE, MESH_PROV_SRV_ADV or MESH_PROXY_SRV_ADV , device_info is invalid.
- **device_info** –**[in]** the device information.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_clean_duplicate_scan_exceptional_list**(*esp_duplicate_scan_exceptional_list_type_t* list_type)

This function is called to clean the duplicate scan exceptional list. This API will delete all device information in the duplicate scan exceptional list.

Parameters **list_type** –**[in]** duplicate scan exceptional list type, the value can be one or more of esp_duplicate_scan_exceptional_list_type_t.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_security_param**(*esp_ble_sm_param_t* param_type, void *value, uint8_t len)

Set a GAP security parameter value. Overrides the default value.

```
                Secure connection is highly recommended to avoid some major
                vulnerabilities like 'Impersonation in the Pin Pairing Protocol'
                (CVE-2020-26555) and 'Authentication of the LE Legacy Pairing
                Protocol'.

                To accept only `secure connection mode`, it is necessary do as␣
↪following:

                1. Set bit `ESP_LE_AUTH_REQ_SC_ONLY` (`param_type` is
                `ESP_BLE_SM_AUTHEN_REQ_MODE`), bit `ESP_LE_AUTH_BOND` and bit
```

(continues on next page)

(continued from previous page)

```
                `ESP_LE_AUTH_REQ_MITM` is optional as required.

                2. Set to `ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE` (`param_
↪type` is
                `ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH`).
```

Parameters
- **param_type** –**[in]** : the type of the param which to be set
- **value** –**[in]** : the param value
- **len** –**[in]** : the length of the param value

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_security_rsp**(*esp_bd_addr_t* bd_addr, bool accept)

Grant security request access.

Parameters
- **bd_addr** –**[in]** : BD address of the peer
- **accept** –**[in]** : accept the security request or not

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_set_encryption**(*esp_bd_addr_t* bd_addr, *esp_ble_sec_act_t* sec_act)

Set a gap parameter value. Use this function to change the default GAP parameter values.

Parameters
- **bd_addr** –**[in]** : the address of the peer device need to encryption
- **sec_act** –**[in]** : This is the security action to indicate what kind of BLE security level is required for the BLE link if the BLE is supported

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_passkey_reply**(*esp_bd_addr_t* bd_addr, bool accept, uint32_t passkey)

Reply the key value to the peer device in the legacy connection stage.

Parameters
- **bd_addr** –**[in]** : BD address of the peer
- **accept** –**[in]** : passkey entry successful or declined.
- **passkey** –**[in]** : passkey value, must be a 6 digit number, can be lead by 0.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_confirm_reply**(*esp_bd_addr_t* bd_addr, bool accept)

Reply the confirm value to the peer device in the secure connection stage.

Parameters
- **bd_addr** –**[in]** : BD address of the peer device
- **accept** –**[in]** : numbers to compare are the same or different.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_remove_bond_device**(*esp_bd_addr_t* bd_addr)

Removes a device from the security database list of peer device. It manages unpairing event while connected.

Parameters **bd_addr** –**[in]** : BD address of the peer device

Returns - ESP_OK : success
- other : failed

int **esp_ble_get_bond_device_num**(void)

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

Returns - >= 0 : bonded devices number.
- ESP_FAIL : failed

esp_err_t **esp_ble_get_bond_device_list** (int *dev_num, *esp_ble_bond_dev_t* *dev_list)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Parameters
- **dev_num** –**[inout]** Indicate the dev_list array(buffer) size as input. If dev_num is large enough, it means the actual number as output. Suggest that dev_num value equal to esp_ble_get_bond_device_num().
- **dev_list** –**[out]** an array(buffer) of *esp_ble_bond_dev_t* type. Use for storing the bonded devices address. The dev_list should be allocated by who call this API.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_oob_req_reply** (*esp_bd_addr_t* bd_addr, uint8_t *TK, uint8_t len)

This function is called to provide the OOB data for SMP in response to ESP_GAP_BLE_OOB_REQ_EVT.

Parameters
- **bd_addr** –**[in]** BD address of the peer device.
- **TK** –**[in]** Temporary Key value, the TK value shall be a 128-bit random number
- **len** –**[in]** length of temporary key, should always be 128-bit

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_sc_oob_req_reply** (*esp_bd_addr_t* bd_addr, uint8_t p_c[16], uint8_t p_r[16])

This function is called to provide the OOB data for SMP in response to ESP_GAP_BLE_SC_OOB_REQ_EVT.

Parameters
- **bd_addr** –**[in]** BD address of the peer device.
- **p_c** –**[in]** Confirmation value, it shall be a 128-bit random number
- **p_r** –**[in]** Randomizer value, it should be a 128-bit random number

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_create_sc_oob_data** (void)

This function is called to create the OOB data for SMP when secure connection.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_disconnect** (*esp_bd_addr_t* remote_device)

This function is to disconnect the physical connection of the peer device gattc may have multiple virtual GATT server connections when multiple app_id registered. esp_ble_gattc_close (esp_gatt_if_t gattc_if, uint16_t conn_id) only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. esp_ble_gap_disconnect(esp_bd_addr_t remote_device) disconnect the physical connection directly.

Parameters **remote_device** –**[in]** : BD address of the peer device
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_get_current_conn_params** (*esp_bd_addr_t* bd_addr, *esp_gap_conn_params_t* *conn_params)

This function is called to read the connection parameters information of the device.

Parameters
- **bd_addr** –**[in]** BD address of the peer device.
- **conn_params** –**[out]** the connection parameters information

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_gap_ble_set_channels** (*esp_gap_ble_channels* channels)

BLE set channels.

Parameters **channels** –**[in]** : The n th such field (in the range 0 to 36) contains the value for the link layer channel index n. 0 means channel n is bad. 1 means channel n is unknown. The most significant bits are reserved and shall be set to 0. At least one channel shall be marked as unknown.

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_gap_ble_set_authorization** (*esp_bd_addr_t* bd_addr, bool authorize)

This function is called to authorized a link after Authentication(MITM protection)

Parameters
- **bd_addr** –**[in]** BD address of the peer device.
- **authorize** –**[out]** Authorized the link or not.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_read_phy** (*esp_bd_addr_t* bd_addr)

This function is used to read the current transmitter PHY and receiver PHY on the connection identified by remote address.

Parameters **bd_addr** –**[in]** : BD address of the peer device

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_preferred_default_phy** (*esp_ble_gap_phy_mask_t* tx_phy_mask, *esp_ble_gap_phy_mask_t* rx_phy_mask)

This function is used to allows the Host to specify its preferred values for the transmitter PHY and receiver PHY to be used for all subsequent connections over the LE transport.

Parameters
- **tx_phy_mask** –**[in]** : indicates the transmitter PHYs that the Host prefers the Controller to use
- **rx_phy_mask** –**[in]** : indicates the receiver PHYs that the Host prefers the Controller to use

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_preferred_phy** (*esp_bd_addr_t* bd_addr, *esp_ble_gap_all_phys_t* all_phys_mask, *esp_ble_gap_phy_mask_t* tx_phy_mask, *esp_ble_gap_phy_mask_t* rx_phy_mask, *esp_ble_gap_prefer_phy_options_t* phy_options)

This function is used to set the PHY preferences for the connection identified by the remote address. The Controller might not be able to make the change (e.g. because the peer does not support the requested PHY) or may decide that the current PHY is preferable.

Parameters
- **bd_addr** –**[in]** : remote address
- **all_phys_mask** –**[in]** : a bit field that allows the Host to specify
- **tx_phy_mask** –**[in]** : a bit field that indicates the transmitter PHYs that the Host prefers the Controller to use
- **rx_phy_mask** –**[in]** : a bit field that indicates the receiver PHYs that the Host prefers the Controller to use
- **phy_options** –**[in]** : a bit field that allows the Host to specify options for PHYs

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_set_rand_addr** (uint8_t instance, *esp_bd_addr_t* rand_addr)

This function is used by the Host to set the random device address specified by the Random_Address parameter.

Parameters
- **instance** –**[in]** : Used to identify an advertising set
- **rand_addr** –**[in]** : Random Device Address

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_set_params**(uint8_t instance, const *esp_ble_gap_ext_adv_params_t* *params)

This function is used by the Host to set the advertising parameters.

Parameters
- **instance** –**[in]** : identifies the advertising set whose parameters are being configured.
- **params** –**[in]** : advertising parameters

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_ext_adv_data_raw**(uint8_t instance, uint16_t length, const uint8_t *data)

This function is used to set the data used in advertising PDUs that have a data field.

Parameters
- **instance** –**[in]** : identifies the advertising set whose data are being configured
- **length** –**[in]** : data length
- **data** –**[in]** : data information

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_ext_scan_rsp_data_raw**(uint8_t instance, uint16_t length, const uint8_t *scan_rsp_data)

This function is used to provide scan response data used in scanning response PDUs.

Parameters
- **instance** –**[in]** : identifies the advertising set whose response data are being configured.
- **length** –**[in]** : responsedata length
- **scan_rsp_data** –**[in]** : response data information

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_start**(uint8_t num_adv, const *esp_ble_gap_ext_adv_t* *ext_adv)

This function is used to request the Controller to enable one or more advertising sets using the advertising sets identified by the instance parameter.

Parameters
- **num_adv** –**[in]** : Number of advertising sets to enable or disable
- **ext_adv** –**[in]** : adv parameters

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_stop**(uint8_t num_adv, const uint8_t *ext_adv_inst)

This function is used to request the Controller to disable one or more advertising sets using the advertising sets identified by the instance parameter.

Parameters
- **num_adv** –**[in]** : Number of advertising sets to enable or disable
- **ext_adv_inst** –**[in]** : ext adv instance

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_set_remove**(uint8_t instance)

This function is used to remove an advertising set from the Controller.

Parameters **instance** –**[in]** : Used to identify an advertising set

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_ext_adv_set_clear**(void)

This function is used to remove all existing advertising sets from the Controller.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_set_params**(uint8_t instance, const *esp_ble_gap_periodic_adv_params_t* *params)

This function is used by the Host to set the parameters for periodic advertising.

Parameters
- **instance** –**[in]** : identifies the advertising set whose periodic advertising parameters are being configured.
- **params** –**[in]** : periodic adv parameters

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_periodic_adv_data_raw**(uint8_t instance, uint16_t length, const uint8_t *data)

This function is used to set the data used in periodic advertising PDUs.

Parameters
- **instance** –**[in]** : identifies the advertising set whose periodic advertising parameters are being configured.
- **length** –**[in]** : the length of periodic data
- **data** –**[in]** : periodic data information

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_start**(uint8_t instance)

This function is used to request the Controller to enable the periodic advertising for the advertising set specified.

Parameters **instance** –**[in]** : Used to identify an advertising set
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_stop**(uint8_t instance)

This function is used to request the Controller to disable the periodic advertising for the advertising set specified.

Parameters **instance** –**[in]** : Used to identify an advertising set
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_ext_scan_params**(const *esp_ble_ext_scan_params_t* *params)

This function is used to set the extended scan parameters to be used on the advertising channels.

Parameters **params** –**[in]** : scan parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_start_ext_scan**(uint32_t duration, uint16_t period)

This function is used to enable scanning.

Parameters
- **duration** –**[in]** Scan duration time, where Time = N * 10 ms. Range: 0x0001 to 0xFFFF.
- **period** –**[in]** Time interval from when the Controller started its last Scan Duration until it begins the subsequent Scan Duration. Time = N * 1.28 sec. Range: 0x0001 to 0xFFFF.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_stop_ext_scan**(void)

This function is used to disable scanning.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_create_sync**(const *esp_ble_gap_periodic_adv_sync_params_t* *params)

This function is used to synchronize with periodic advertising from an advertiser and begin receiving periodic advertising packets.

Parameters **params** –**[in]** : sync parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_sync_cancel**(void)

This function is used to cancel the LE_Periodic_Advertising_Create_Sync command while it is pending.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_sync_terminate**(uint16_t sync_handle)

This function is used to stop reception of the periodic advertising identified by the Sync Handle parameter.

Parameters **sync_handle** –**[in]** : identify the periodic advertiser
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_add_dev_to_list**(*esp_ble_addr_type_t* addr_type, *esp_bd_addr_t* addr, uint8_t sid)

This function is used to add a single device to the Periodic Advertiser list stored in the Controller.

Parameters
- **addr_type** –**[in]** : address type
- **addr** –**[in]** : Device Address
- **sid** –**[in]** : Advertising SID subfield in the ADI field used to identify the Periodic Advertising

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_remove_dev_from_list**(*esp_ble_addr_type_t* addr_type, *esp_bd_addr_t* addr, uint8_t sid)

This function is used to remove one device from the list of Periodic Advertisers stored in the Controller. Removals from the Periodic Advertisers List take effect immediately.

Parameters
- **addr_type** –**[in]** : address type
- **addr** –**[in]** : Device Address
- **sid** –**[in]** : Advertising SID subfield in the ADI field used to identify the Periodic Advertising

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_periodic_adv_clear_dev**(void)

This function is used to remove all devices from the list of Periodic Advertisers in the Controller.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_prefer_ext_connect_params_set**(*esp_bd_addr_t* addr, *esp_ble_gap_phy_mask_t* phy_mask, const *esp_ble_gap_conn_params_t* *phy_1m_conn_params, const *esp_ble_gap_conn_params_t* *phy_2m_conn_params, const *esp_ble_gap_conn_params_t* *phy_coded_conn_params)

This function is used to set aux connection parameters.

Parameters

- **addr** –**[in]** : device address
- **phy_mask** –**[in]** : indicates the PHY(s) on which the advertising packets should be received on the primary advertising channel and the PHYs for which connection parameters have been specified.
- **phy_1m_conn_params** –**[in]** : Scan connectable advertisements on the LE 1M PHY. Connection parameters for the LE 1M PHY are provided.
- **phy_2m_conn_params** –**[in]** : Connection parameters for the LE 2M PHY are provided.
- **phy_coded_conn_params** –**[in]** : Scan connectable advertisements on the LE Coded PHY. Connection parameters for the LE Coded PHY are provided.

Returns - ESP_OK : success

- other : failed

esp_err_t **esp_ble_gap_periodic_adv_recv_enable**(uint16_t sync_handle, uint8_t enable)

This function is used to set periodic advertising receive enable.

Parameters

- **sync_handle** –**[in]** : Handle of periodic advertising sync
- **enable** –**[in]** : Determines whether reporting and duplicate filtering are enabled or disabled

Returns - ESP_OK : success

- other : failed

esp_err_t **esp_ble_gap_periodic_adv_sync_trans**(*esp_bd_addr_t* addr, uint16_t service_data, uint16_t sync_handle)

This function is used to transfer periodic advertising sync.

Parameters

- **addr** –**[in]** : Peer device address
- **service_data** –**[in]** : Service data used by Host
- **sync_handle** –**[in]** : Handle of periodic advertising sync

Returns - ESP_OK : success

- other : failed

esp_err_t **esp_ble_gap_periodic_adv_set_info_trans**(*esp_bd_addr_t* addr, uint16_t service_data, uint8_t adv_handle)

This function is used to transfer periodic advertising set info.

Parameters

- **addr** –**[in]** : Peer device address
- **service_data** –**[in]** : Service data used by Host
- **adv_handle** –**[in]** : Handle of advertising set

Returns - ESP_OK : success

- other : failed

esp_err_t **esp_ble_gap_set_periodic_adv_sync_trans_params**(*esp_bd_addr_t* addr, const *esp_ble_gap_past_params_t* *params)

This function is used to set periodic advertising sync transfer params.

Parameters
- **addr** –**[in]** : Peer device address
- **params** –**[in]** : Params of periodic advertising sync transfer

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_dtm_tx_start**(const *esp_ble_dtm_tx_t* *tx_params)

This function is used to start a test where the DUT generates reference packets at a fixed interval.

Parameters **tx_params** –**[in]** : DTM Transmitter parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_dtm_rx_start**(const *esp_ble_dtm_rx_t* *rx_params)

This function is used to start a test where the DUT receives test reference packets at a fixed interval.

Parameters **rx_params** –**[in]** : DTM Receiver parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_dtm_enh_tx_start**(const *esp_ble_dtm_enh_tx_t* *tx_params)

This function is used to start a test where the DUT generates reference packets at a fixed interval.

Parameters **tx_params** –**[in]** : DTM Transmitter parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_dtm_enh_rx_start**(const *esp_ble_dtm_enh_rx_t* *rx_params)

This function is used to start a test where the DUT receives test reference packets at a fixed interval.

Parameters **rx_params** –**[in]** : DTM Receiver parameters
Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_dtm_stop**(void)

This function is used to stop any test which is in progress.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_clear_advertising**(void)

This function is used to clear legacy advertising.

Returns - ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_vendor_command_send**(*esp_ble_vendor_cmd_params_t* *vendor_cmd_param)

This function is called to send vendor hci command.

Parameters **vendor_cmd_param** –**[in]** vendor hci command parameters
Returns
- ESP_OK : success
- other : failed

Unions

union **esp_ble_key_value_t**

#include <esp_gap_ble_api.h> union type of the security key value

Public Members

esp_ble_penc_keys_t **penc_key**

received peer encryption key

esp_ble_pcsrk_keys_t **pcsrk_key**

received peer device SRK

esp_ble_pid_keys_t **pid_key**

peer device ID key

esp_ble_lenc_keys_t **lenc_key**

local encryption reproduction keys LTK = = d1(ER,DIV,0)

esp_ble_lcsrk_keys **lcsrk_key**

local device CSRK = d1(ER,DIV,1)

union **esp_ble_sec_t**

#include <esp_gap_ble_api.h> union associated with ble security

Public Members

esp_ble_sec_key_notif_t **key_notif**

passkey notification

esp_ble_sec_req_t **ble_req**

BLE SMP related request

esp_ble_key_t **ble_key**

BLE SMP keys used when pairing

esp_ble_local_id_keys_t **ble_id_keys**

BLE IR event

esp_ble_local_oob_data_t **oob_data**

BLE SMP secure connection OOB data

esp_ble_auth_cmpl_t **auth_cmpl**

Authentication complete indication.

union **esp_ble_gap_cb_param_t**

#include <esp_gap_ble_api.h> Gap callback parameters union.

Public Members

struct *esp_ble_gap_cb_param_t*::*ble_get_dev_name_cmpl_evt_param* **get_dev_name_cmpl**

Event parameter of ESP_GAP_BLE_GET_DEV_NAME_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_data_cmpl_evt_param* **adv_data_cmpl**

Event parameter of ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_rsp_data_cmpl_evt_param* **scan_rsp_data_cmpl**

Event parameter of ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_param_cmpl_evt_param* **scan_param_cmpl**

Event parameter of ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_result_evt_param* **scan_rst**

Event parameter of ESP_GAP_BLE_SCAN_RESULT_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_data_raw_cmpl_evt_param* **adv_data_raw_cmpl**

Event parameter of ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_rsp_data_raw_cmpl_evt_param* **scan_rsp_data_raw_cmpl**

Event parameter of ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_start_cmpl_evt_param* **adv_start_cmpl**

Event parameter of ESP_GAP_BLE_ADV_START_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_start_cmpl_evt_param* **scan_start_cmpl**

Event parameter of ESP_GAP_BLE_SCAN_START_COMPLETE_EVT

esp_ble_sec_t **ble_security**

ble gap security union type

struct *esp_ble_gap_cb_param_t*::*ble_scan_stop_cmpl_evt_param* **scan_stop_cmpl**

Event parameter of ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_stop_cmpl_evt_param* **adv_stop_cmpl**

Event parameter of ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_clear_cmpl_evt_param* **adv_clear_cmpl**

Event parameter of ESP_GAP_BLE_ADV_CLEAR_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_rand_cmpl_evt_param* **set_rand_addr_cmpl**

Event parameter of ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT

struct *esp_ble_gap_cb_param_t*::*ble_update_conn_params_evt_param* **update_conn_params**

Event parameter of ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT

struct *esp_ble_gap_cb_param_t*::*ble_pkt_data_length_cmpl_evt_param* **pkt_data_length_cmpl**

Event parameter of ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_local_privacy_cmpl_evt_param* **local_privacy_cmpl**

Event parameter of ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_remove_bond_dev_cmpl_evt_param* **remove_bond_dev_cmpl**

Event parameter of ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_clear_bond_dev_cmpl_evt_param* **clear_bond_dev_cmpl**

Event parameter of ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_get_bond_dev_cmpl_evt_param* **get_bond_dev_cmpl**

Event parameter of ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_read_rssi_cmpl_evt_param* **read_rssi_cmpl**

Event parameter of ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_update_whitelist_cmpl_evt_param* **update_whitelist_cmpl**

Event parameter of ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_update_duplicate_exceptional_list_cmpl_evt_param* **update_duplicate_exceptional_list_cmpl**

Event parameter of ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_channels_evt_param* **ble_set_channels**

Event parameter of ESP_GAP_BLE_SET_CHANNELS_EVT

struct *esp_ble_gap_cb_param_t*::*ble_read_phy_cmpl_evt_param* **read_phy**

Event parameter of ESP_GAP_BLE_READ_PHY_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_perf_def_phy_cmpl_evt_param* **set_perf_def_phy**

Event parameter of ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_perf_phy_cmpl_evt_param* **set_perf_phy**

Event parameter of ESP_GAP_BLE_SET_PREFERRED_PHY_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_set_rand_addr_cmpl_evt_param* **ext_adv_set_rand_addr**

Event parameter of ESP_GAP_BLE_EXT_ADV_SET_RAND_ADDR_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_set_params_cmpl_evt_param* **ext_adv_set_params**

Event parameter of ESP_GAP_BLE_EXT_ADV_SET_PARAMS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_data_set_cmpl_evt_param* **ext_adv_data_set**

Event parameter of ESP_GAP_BLE_EXT_ADV_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_scan_rsp_set_cmpl_evt_param* **scan_rsp_set**

Event parameter of ESP_GAP_BLE_EXT_SCAN_RSP_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_start_cmpl_evt_param* **ext_adv_start**

Event parameter of ESP_GAP_BLE_EXT_ADV_START_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_stop_cmpl_evt_param* **ext_adv_stop**

Event parameter of ESP_GAP_BLE_EXT_ADV_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_set_remove_cmpl_evt_param* **ext_adv_remove**

Event parameter of ESP_GAP_BLE_EXT_ADV_SET_REMOVE_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_set_clear_cmpl_evt_param* **ext_adv_clear**

Event parameter of ESP_GAP_BLE_EXT_ADV_SET_CLEAR_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_set_params_cmpl_param* **peroid_adv_set_params**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SET_PARAMS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_data_set_cmpl_param* **period_adv_data_set**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_start_cmpl_param* **period_adv_start**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_START_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_stop_cmpl_param* **period_adv_stop**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_create_sync_cmpl_param* **period_adv_create_sync**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_CREATE_SYNC_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_sync_cancel_cmpl_param* **period_adv_sync_cancel**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_CANCEL_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_sync_terminate_cmpl_param* **period_adv_sync_term**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_TERMINATE_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_add_dev_cmpl_param* **period_adv_add_dev**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_ADD_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_remove_dev_cmpl_param* **period_adv_remove_dev**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_REMOVE_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_period_adv_clear_dev_cmpl_param* **period_adv_clear_dev**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_CLEAR_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_ext_scan_params_cmpl_param* **set_ext_scan_params**

Event parameter of ESP_GAP_BLE_SET_EXT_SCAN_PARAMS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_scan_start_cmpl_param* **ext_scan_start**

Event parameter of ESP_GAP_BLE_EXT_SCAN_START_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_scan_stop_cmpl_param* **ext_scan_stop**

Event parameter of ESP_GAP_BLE_EXT_SCAN_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_conn_params_set_cmpl_param* **ext_conn_params_set**

Event parameter of ESP_GAP_BLE_PREFER_EXT_CONN_PARAMS_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_adv_terminate_param* **adv_terminate**

Event parameter of ESP_GAP_BLE_ADV_TERMINATED_EVT

struct *esp_ble_gap_cb_param_t*::*ble_scan_req_received_param* **scan_req_received**

Event parameter of ESP_GAP_BLE_SCAN_REQ_RECEIVED_EVT

struct *esp_ble_gap_cb_param_t*::*ble_channel_sel_alg_param* **channel_sel_alg**

Event parameter of ESP_GAP_BLE_CHANNEL_SELECT_ALGORITHM_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_sync_lost_param* **periodic_adv_sync_lost**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_LOST_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_sync_estab_param* **periodic_adv_sync_estab**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_ESTAB_EVT

struct *esp_ble_gap_cb_param_t*::*ble_phy_update_cmpl_param* **phy_update**

Event parameter of ESP_GAP_BLE_PHY_UPDATE_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_ext_adv_report_param* **ext_adv_report**

Event parameter of ESP_GAP_BLE_EXT_ADV_REPORT_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_report_param* **period_adv_report**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_REPORT_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_recv_enable_cmpl_param* **period_adv_recv_enable**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_RECV_ENABLE_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_sync_trans_cmpl_param* **period_adv_sync_trans**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_set_info_trans_cmpl_param* **period_adv_set_info_trans**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SET_INFO_TRANS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_set_past_params_cmpl_param* **set_past_params**

Event parameter of ESP_GAP_BLE_SET_PAST_PARAMS_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t*::*ble_periodic_adv_sync_trans_recv_param* **past_received**

Event parameter of ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_RECV_EVT

struct *esp_ble_gap_cb_param_t*::*ble_dtm_state_update_evt_param* **dtm_state_update**

Event parameter of ESP_GAP_BLE_DTM_TEST_UPDATE_EVT

struct *esp_ble_gap_cb_param_t*::*vendor_cmd_cmpl_evt_param* **vendor_cmd_cmpl**

Event parameter of ESP_GAP_BLE_VENDOR_CMD_COMPLETE_EVT

struct **ble_adv_clear_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_CLEAR_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate adv clear operation success status

struct **ble_adv_data_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set advertising data operation success status

struct **ble_adv_data_raw_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set raw advertising data operation success status

struct **ble_adv_start_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate advertising start operation success status

struct **ble_adv_stop_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate adv stop operation success status

struct **ble_adv_terminate_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_TERMINATED_EVT.

Public Members

uint8_t **status**

Indicate adv terminate status

uint8_t **adv_instance**

extend advertising handle

uint16_t **conn_idx**

connection index

uint8_t **completed_event**

the number of completed extend advertising events

struct **ble_channel_sel_alg_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_CHANNEL_SELECT_ALGORITHM_EVT.

Public Members

uint16_t **conn_handle**

connection handle

uint8_t **channel_sel_alg**

channel selection algorithm

struct **ble_clear_bond_dev_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the clear bond device operation success status

struct **ble_dtm_state_update_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_DTM_TEST_UPDATE_EVT.

Public Members

esp_bt_status_t **status**

Indicate DTM operation success status

esp_ble_dtm_update_evt_t **update_evt**

DTM state change event, 0x00: DTM TX start, 0x01: DTM RX start, 0x02:DTM end

uint16_t **num_of_pkt**

number of packets received, only valid if update_evt is DTM_TEST_STOP_EVT and shall be reported as 0 for a transmitter

struct **ble_ext_adv_data_set_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising data set status

struct **ble_ext_adv_report_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_REPORT_EVT.

Public Members

esp_ble_gap_ext_adv_report_t **params**

extend advertising report parameters

struct **ble_ext_adv_scan_rsp_set_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_SCAN_RSP_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising scan response data set status

struct **ble_ext_adv_set_clear_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_SET_CLEAR_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate advertising stop operation success status

struct **ble_ext_adv_set_params_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_SET_PARAMS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising parameters set status

struct **ble_ext_adv_set_rand_addr_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_SET_RAND_ADDR_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising random address set status

struct **ble_ext_adv_set_remove_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_SET_REMOVE_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate advertising stop operation success status

struct **ble_ext_adv_start_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate advertising start operation success status

struct **ble_ext_adv_stop_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_ADV_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate advertising stop operation success status

struct **ble_ext_conn_params_set_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PREFER_EXT_CONN_PARAMS_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend connection parameters set status

struct **ble_ext_scan_start_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_SCAN_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising start status

struct **ble_ext_scan_stop_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_EXT_SCAN_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising stop status

struct **ble_get_bond_dev_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the get bond device operation success status

uint8_t **dev_num**

Indicate the get number device in the bond list

esp_ble_bond_dev_t ***bond_dev**

the pointer to the bond device Structure

struct **ble_get_dev_name_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_GET_DEV_NAME_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the get device name success status

char ***name**

Name of bluetooth device

struct **ble_local_privacy_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set local privacy operation success status

struct **ble_period_adv_add_dev_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_ADD_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising device list add status

struct **ble_period_adv_clear_dev_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_CLEAR_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising device list clean status

struct **ble_period_adv_create_sync_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_CREATE_SYNC_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising create sync status

struct **ble_period_adv_remove_dev_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_REMOVE_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising device list remove status

struct **ble_period_adv_sync_cancel_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_CANCEL_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising sync cancel status

struct **ble_period_adv_sync_terminate_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_TERMINATE_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising sync terminate status

struct **ble_periodic_adv_data_set_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising data set status

struct **ble_periodic_adv_recv_enable_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_RECV_ENABLE_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Set periodic advertising receive enable status

struct **ble_periodic_adv_report_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_REPORT_EVT.

Public Members

esp_ble_gap_periodic_adv_report_t **params**

periodic advertising report parameters

struct **ble_periodic_adv_set_info_trans_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SET_INFO_TRANS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Periodic advertising set info transfer status

esp_bd_addr_t **bda**

The remote device address

struct **ble_periodic_adv_set_params_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SET_PARAMS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertisingparameters set status

struct **ble_periodic_adv_start_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising start status

struct **ble_periodic_adv_stop_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate periodic advertising stop status

struct **ble_periodic_adv_sync_estab_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_ESTAB_EVT.

Public Members

uint8_t **status**

periodic advertising sync status

uint16_t **sync_handle**

periodic advertising sync handle

uint8_t **sid**

periodic advertising sid

esp_ble_addr_type_t **adv_addr_type**

periodic advertising address type

esp_bd_addr_t **adv_addr**

periodic advertising address

esp_ble_gap_phy_t **adv_phy**

periodic advertising phy type

uint16_t **period_adv_interval**

periodic advertising interval

uint8_t **adv_clk_accuracy**

periodic advertising clock accuracy

struct **ble_periodic_adv_sync_lost_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_LOST_EVT.

Public Members

uint16_t **sync_handle**

sync handle

struct **ble_periodic_adv_sync_trans_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Periodic advertising sync transfer status

esp_bd_addr_t **bda**

The remote device address

struct **ble_periodic_adv_sync_trans_recv_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_RECV_EVT.

Public Members

esp_bt_status_t **status**

Periodic advertising sync transfer received status

esp_bd_addr_t **bda**

The remote device address

uint16_t **service_data**

The value provided by the peer device

uint16_t **sync_handle**

Periodic advertising sync handle

uint8_t **adv_sid**

Periodic advertising set id

uint8_t **adv_addr_type**

Periodic advertiser address type

esp_bd_addr_t **adv_addr**

Periodic advertiser address

esp_ble_gap_phy_t **adv_phy**

Periodic advertising PHY

uint16_t **adv_interval**

Periodic advertising interval

uint8_t **adv_clk_accuracy**

Periodic advertising clock accuracy

struct **ble_phy_update_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_PHY_UPDATE_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

phy update status

esp_bd_addr_t **bda**

address

esp_ble_gap_phy_t **tx_phy**

tx phy type

esp_ble_gap_phy_t **rx_phy**

rx phy type

struct **ble_pkt_data_length_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set pkt data length operation success status

esp_ble_pkt_data_length_params_t **params**

pkt data length value

struct **ble_read_phy_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_READ_PHY_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

read phy complete status

esp_bd_addr_t **bda**

read phy address

esp_ble_gap_phy_t **tx_phy**

tx phy type

esp_ble_gap_phy_t **rx_phy**

rx phy type

struct **ble_read_rssi_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the read adv tx power operation success status

int8_t **rssi**

The ble remote device rssi value, the range is from -127 to 20, the unit is dbm, if the RSSI cannot be read, the RSSI metric shall be set to 127.

esp_bd_addr_t **remote_addr**

The remote device address

struct **ble_remove_bond_dev_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the remove bond device operation success status

esp_bd_addr_t **bd_addr**

The device address which has been remove from the bond list

struct **ble_scan_param_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set scan param operation success status

struct **ble_scan_req_received_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_REQ_RECEIVED_EVT.

Public Members

uint8_t **adv_instance**

extend advertising handle

esp_ble_addr_type_t **scan_addr_type**

scanner address type

esp_bd_addr_t **scan_addr**

scanner address

struct **ble_scan_result_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RESULT_EVT.

Public Members

esp_gap_search_evt_t **search_evt**

Search event type

esp_bd_addr_t **bda**

Bluetooth device address which has been searched

esp_bt_dev_type_t **dev_type**

Device type

esp_ble_addr_type_t **ble_addr_type**

Ble device address type

esp_ble_evt_type_t **ble_evt_type**

Ble scan result event type

int **rssi**

Searched device’s RSSI

uint8_t **ble_adv**[ESP_BLE_ADV_DATA_LEN_MAX + ESP_BLE_SCAN_RSP_DATA_LEN_MAX]

Received EIR

int **flag**

Advertising data flag bit

int **num_resps**

Scan result number

uint8_t **adv_data_len**

Adv data length

uint8_t **scan_rsp_len**

Scan response length

uint32_t **num_dis**

The number of discard packets

struct **ble_scan_rsp_data_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set scan response data operation success status

struct **ble_scan_rsp_data_raw_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set raw advertising data operation success status

struct **ble_scan_start_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate scan start operation success status

struct **ble_scan_stop_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate scan stop operation success status

struct **ble_set_channels_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_CHANNELS_EVT.

Public Members

esp_bt_status_t **stat**

BLE set channel status

struct **ble_set_ext_scan_params_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_EXT_SCAN_PARAMS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate extend advertising parameters set status

struct **ble_set_past_params_cmpl_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_PAST_PARAMS_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Set periodic advertising sync transfer params status

esp_bd_addr_t **bda**

The remote device address

struct **ble_set_perf_def_phy_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate perf default phy set status

struct **ble_set_perf_phy_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_PREFERRED_PHY_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate perf phy set status

struct **ble_set_rand_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT.

Public Members

esp_bt_status_t **status**

Indicate set static rand address operation success status

struct **ble_update_conn_params_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT.

Public Members

esp_bt_status_t **status**

Indicate update connection parameters success status

esp_bd_addr_t **bda**

Bluetooth device address

uint16_t **min_int**

Min connection interval

uint16_t **max_int**

Max connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t **conn_int**

Current connection interval

uint16_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec

struct **ble_update_duplicate_exceptional_list_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate update duplicate scan exceptional list operation success status

uint8_t **subcode**

Define in esp_bt_duplicate_exceptional_subcode_type_t

uint16_t **length**

The length of device_info

esp_duplicate_info_t **device_info**

device information, when subcode is ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_CLEAN, the value is invalid

struct **ble_update_whitelist_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the add or remove whitelist operation success status

esp_ble_wl_operation_t **wl_operation**

The value is ESP_BLE_WHITELIST_ADD if add address to whitelist operation success, ESP_BLE_WHITELIST_REMOVE if remove address from the whitelist operation success

struct **vendor_cmd_cmpl_evt_param**

#include <esp_gap_ble_api.h> ESP_GAP_BLE_VENDOR_CMD_COMPLETE_EVT.

Public Members

uint16_t **opcode**

vendor hci command opcode

uint16_t **param_len**

The length of parameter buffer

uint8_t ***p_param_buf**

The point of parameter buffer

Structures

struct **esp_ble_vendor_cmd_params_t**

Vendor HCI command parameters.

Public Members

uint16_t **opcode**

vendor hci command opcode

uint8_t **param_len**

the length of parameter

uint8_t ***p_param_buf**

the point of parameter buffer

struct **esp_ble_dtm_tx_t**

DTM TX parameters.

Public Members

uint8_t **tx_channel**

channel for sending test data, tx_channel = (Frequency -2402)/2, tx_channel range:0x00-0x27, Frequency range: 2402 MHz to 2480 MHz

uint8_t **len_of_data**

length in bytes of payload data in each packet

esp_ble_dtm_pkt_payload_t **pkt_payload**

packet payload type. value range: 0x00-0x07

struct **esp_ble_dtm_rx_t**

DTM RX parameters.

Public Members

uint8_t **rx_channel**

channel for test data reception, rx_channel = (Frequency -2402)/2, tx_channel range:0x00-0x27, Frequency range: 2402 MHz to 2480 MHz

struct **esp_ble_adv_params_t**

Advertising parameters.

Public Members

uint16_t **adv_int_min**

Minimum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N * 0.625 msec Time Range: 20 ms to 10.24 sec

uint16_t **adv_int_max**

Maximum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N * 0.625 msec Time Range: 20 ms to 10.24 sec Advertising max interval

esp_ble_adv_type_t **adv_type**

Advertising type

esp_ble_addr_type_t **own_addr_type**

Owner bluetooth device address type

esp_bd_addr_t **peer_addr**

Peer device bluetooth device address

esp_ble_addr_type_t **peer_addr_type**

Peer device bluetooth device address type, only support public address type and random address type

esp_ble_adv_channel_t **channel_map**

Advertising channel map

esp_ble_adv_filter_t **adv_filter_policy**

Advertising filter policy

struct **esp_ble_adv_data_t**

Advertising data content, according to “Supplement to the Bluetooth Core Specification”.

Public Members

bool **set_scan_rsp**

Set this advertising data as scan response or not

bool **include_name**

Advertising data include device name or not

bool **include_txpower**

Advertising data include TX power

int **min_interval**

Advertising data show slave preferred connection min interval. The connection interval in the following manner: connIntervalmin = Conn_Interval_Min * 1.25 ms Conn_Interval_Min range: 0x0006 to 0x0C80 Value of 0xFFFF indicates no specific minimum. Values not defined above are reserved for future use.

int **max_interval**

Advertising data show slave preferred connection max interval. The connection interval in the following manner: connIntervalmax = Conn_Interval_Max * 1.25 ms Conn_Interval_Max range: 0x0006 to 0x0C80 Conn_Interval_Max shall be equal to or greater than the Conn_Interval_Min. Value of 0xFFFF indicates no specific maximum. Values not defined above are reserved for future use.

int **appearance**

External appearance of device

uint16_t **manufacturer_len**

Manufacturer data length

uint8_t ***p_manufacturer_data**

Manufacturer data point

uint16_t **service_data_len**

Service data length

uint8_t ***p_service_data**

Service data point

uint16_t **service_uuid_len**

Service uuid length

uint8_t ***p_service_uuid**

Service uuid array point

uint8_t **flag**

Advertising flag of discovery mode, see BLE_ADV_DATA_FLAG detail

struct **esp_ble_scan_params_t**

Ble scan parameters.

Public Members

esp_ble_scan_type_t **scan_type**

Scan type

esp_ble_addr_type_t **own_addr_type**

Owner address type

esp_ble_scan_filter_t **scan_filter_policy**

Scan filter policy

uint16_t **scan_interval**

Scan interval. This is defined as the time interval from when the Controller started its last LE scan until it begins the subsequent LE scan. Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N * 0.625 msec Time Range: 2.5 msec to 10.24 seconds

uint16_t **scan_window**

Scan window. The duration of the LE scan. LE_Scan_Window shall be less than or equal to LE_Scan_Interval Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N * 0.625 msec Time Range: 2.5 msec to 10240 msec

esp_ble_scan_duplicate_t **scan_duplicate**

The Scan_Duplicates parameter controls whether the Link Layer should filter out duplicate advertising reports (BLE_SCAN_DUPLICATE_ENABLE) to the Host, or if the Link Layer should generate advertising reports for each packet received

struct **esp_gap_conn_params_t**

connection parameters information

Public Members

uint16_t **interval**

connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct **esp_ble_conn_update_params_t**

Connection update parameters.

Public Members

esp_bd_addr_t **bda**

Bluetooth device address

uint16_t **min_int**

Min connection interval

uint16_t **max_int**

Max connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct **esp_ble_pkt_data_length_params_t**

BLE pkt date length keys.

Public Members

uint16_t **rx_len**

pkt rx data length value

uint16_t **tx_len**

pkt tx data length value

struct **esp_ble_penc_keys_t**

BLE encryption keys.

Public Members

esp_bt_octet16_t **ltk**

The long term key

esp_bt_octet8_t **rand**

The random number

uint16_t **ediv**

The ediv value

uint8_t **sec_level**

The security level of the security link

uint8_t **key_size**

The key size(7~16) of the security link

struct **esp_ble_pcsrk_keys_t**

BLE CSRK keys.

Public Members

uint32_t **counter**

The counter

esp_bt_octet16_t **csrk**

The csrk key

uint8_t **sec_level**

The security level

struct **esp_ble_pid_keys_t**

BLE pid keys.

Public Members

esp_bt_octet16_t **irk**

The irk value

esp_ble_addr_type_t **addr_type**

The address type

esp_bd_addr_t **static_addr**

The static address

struct **esp_ble_lenc_keys_t**

BLE Encryption reproduction keys.

Public Members

esp_bt_octet16_t **ltk**

The long term key

uint16_t **div**

The div value

uint8_t **key_size**

The key size of the security link

uint8_t **sec_level**

The security level of the security link

struct **esp_ble_lcsrk_keys**

BLE SRK keys.

Public Members

uint32_t **counter**

The counter value

uint16_t **div**

The div value

uint8_t **sec_level**

The security level of the security link

esp_bt_octet16_t **csrk**

The csrk key value

struct **esp_ble_sec_key_notif_t**

Structure associated with ESP_KEY_NOTIF_EVT.

Public Members

esp_bd_addr_t **bd_addr**

peer address

uint32_t **passkey**

the numeric value for comparison. If just_works, do not show this number to UI

struct **esp_ble_sec_req_t**

Structure of the security request.

Public Members

esp_bd_addr_t **bd_addr**

peer address

struct **esp_ble_bond_key_info_t**

struct type of the bond key information value

Public Members

esp_ble_key_mask_t **key_mask**

the key mask to indicate witch key is present

esp_ble_penc_keys_t **penc_key**

received peer encryption key

esp_ble_pcsrk_keys_t **pcsrk_key**

received peer device SRK

esp_ble_pid_keys_t **pid_key**

peer device ID key

struct **esp_ble_bond_dev_t**

struct type of the bond device value

Public Members

esp_bd_addr_t **bd_addr**

peer address

esp_ble_bond_key_info_t **bond_key**

the bond key information

struct **esp_ble_key_t**

union type of the security key value

Public Members

esp_bd_addr_t **bd_addr**

peer address

esp_ble_key_type_t **key_type**

key type of the security link

esp_ble_key_value_t **p_key_value**

the pointer to the key value

struct **esp_ble_local_id_keys_t**

structure type of the ble local id keys value

Public Members

esp_bt_octet16_t **ir**

the 16 bits of the ir value

esp_bt_octet16_t **irk**

the 16 bits of the ir key value

esp_bt_octet16_t **dhk**

the 16 bits of the dh key value

struct **esp_ble_local_oob_data_t**

structure type of the ble local oob data value

Public Members

esp_bt_octet16_t **oob_c**

the 128 bits of confirmation value

esp_bt_octet16_t **oob_r**

the 128 bits of randomizer value

struct **esp_ble_auth_cmpl_t**

Structure associated with ESP_AUTH_CMPL_EVT.

Public Members

esp_bd_addr_t **bd_addr**

BD address peer device.

bool **key_present**

Valid link key value in key element

esp_link_key **key**

Link key associated with peer device.

uint8_t **key_type**

The type of Link Key

bool **success**

TRUE of authentication succeeded, FALSE if failed.

uint8_t **fail_reason**

The HCI reason/error code for when success=FALSE

esp_ble_addr_type_t **addr_type**

Peer device address type

esp_bt_dev_type_t **dev_type**

Device type

esp_ble_auth_req_t **auth_mode**

authentication mode

struct **esp_ble_gap_ext_adv_params_t**

ext adv parameters

Public Members

esp_ble_ext_adv_type_mask_t **type**

ext adv type

uint32_t **interval_min**

ext adv minimum interval

uint32_t **interval_max**

ext adv maximum interval

esp_ble_adv_channel_t **channel_map**

ext adv channel map

esp_ble_addr_type_t **own_addr_type**

ext adv own address type

esp_ble_addr_type_t **peer_addr_type**

ext adv peer address type

esp_bd_addr_t **peer_addr**

ext adv peer address

esp_ble_adv_filter_t **filter_policy**

ext adv filter policy

int8_t **tx_power**

ext adv tx power

esp_ble_gap_pri_phy_t **primary_phy**

ext adv primary phy

uint8_t **max_skip**

ext adv maximum skip

esp_ble_gap_phy_t **secondary_phy**

ext adv secondary phy

uint8_t **sid**

ext adv sid

bool **scan_req_notif**

ext adv scan request event notify

struct **esp_ble_ext_scan_cfg_t**

ext scan config

Public Members

esp_ble_scan_type_t **scan_type**

ext scan type

uint16_t **scan_interval**

ext scan interval

uint16_t **scan_window**

ext scan window

struct **esp_ble_ext_scan_params_t**

ext scan parameters

Public Members

esp_ble_addr_type_t **own_addr_type**

ext scan own address type

esp_ble_scan_filter_t **filter_policy**

ext scan filter policy

esp_ble_scan_duplicate_t **scan_duplicate**

ext scan duplicate scan

esp_ble_ext_scan_cfg_mask_t **cfg_mask**

ext scan config mask

esp_ble_ext_scan_cfg_t **uncoded_cfg**

ext scan uncoded config parameters

esp_ble_ext_scan_cfg_t **coded_cfg**

ext scan coded config parameters

struct **esp_ble_gap_conn_params_t**

create extend connection parameters

Public Members

uint16_t **scan_interval**

init scan interval

uint16_t **scan_window**

init scan window

uint16_t **interval_min**

minimum interval

uint16_t **interval_max**

maximum interval

uint16_t **latency**

ext scan type

uint16_t **supervision_timeout**

connection supervision timeout

uint16_t **min_ce_len**

minimum ce length

uint16_t **max_ce_len**

maximum ce length

struct **esp_ble_gap_ext_adv_t**

extend adv enable parameters

Public Members

uint8_t **instance**

advertising handle

int **duration**

advertising duration

int **max_events**

maximum number of extended advertising events

struct **esp_ble_gap_periodic_adv_params_t**

periodic adv parameters

Public Members

uint16_t **interval_min**

periodic advertising minimum interval

uint16_t **interval_max**

periodic advertising maximum interval

uint8_t **properties**

periodic advertising properties

struct **esp_ble_gap_periodic_adv_sync_params_t**

periodic adv sync parameters

Public Members

esp_ble_gap_sync_t **filter_policy**

Configures the filter policy for periodic advertising sync: 0: Use Advertising SID, Advertiser Address Type, and Advertiser Address parameters to determine the advertiser to listen to. 1: Use the Periodic Advertiser List to determine the advertiser to listen to.

uint8_t **sid**

SID of the periodic advertising

esp_ble_addr_type_t **addr_type**

Address type of the periodic advertising

esp_bd_addr_t **addr**

Address of the periodic advertising

uint16_t **skip**

Maximum number of periodic advertising events that can be skipped

uint16_t **sync_timeout**

Synchronization timeout

struct **esp_ble_gap_ext_adv_report_t**

extend adv report parameters

Public Members

esp_ble_gap_adv_type_t **event_type**

extend advertising type

uint8_t **addr_type**

extend advertising address type

esp_bd_addr_t **addr**

extend advertising address

esp_ble_gap_pri_phy_t **primary_phy**

extend advertising primary phy

esp_ble_gap_phy_t **secondly_phy**

extend advertising secondary phy

uint8_t **sid**

extend advertising sid

uint8_t **tx_power**

extend advertising tx power

int8_t **rssi**

extend advertising rssi

uint16_t **per_adv_interval**
: periodic advertising interval

uint8_t **dir_addr_type**
: direct address type

esp_bd_addr_t **dir_addr**
: direct address

esp_ble_gap_ext_adv_data_status_t **data_status**
: data type

uint8_t **adv_data_len**
: extend advertising data length

uint8_t **adv_data**[251]
: extend advertising data

struct **esp_ble_gap_periodic_adv_report_t**
: periodic adv report parameters

Public Members

uint16_t **sync_handle**
: periodic advertising train handle

uint8_t **tx_power**
: periodic advertising tx power

int8_t **rssi**
: periodic advertising rssi

esp_ble_gap_ext_adv_data_status_t **data_status**
: periodic advertising data type

uint8_t **data_length**
: periodic advertising data length

uint8_t **data**[251]
: periodic advertising data

struct **esp_ble_gap_periodic_adv_sync_estab_t**
: perodic adv sync establish parameters

Public Members

uint8_t **status**

periodic advertising sync status

uint16_t **sync_handle**

periodic advertising train handle

uint8_t **sid**

periodic advertising sid

esp_ble_addr_type_t **addr_type**

periodic advertising address type

esp_bd_addr_t **adv_addr**

periodic advertising address

esp_ble_gap_phy_t **adv_phy**

periodic advertising adv phy type

uint16_t **period_adv_interval**

periodic advertising interval

uint8_t **adv_clk_accuracy**

periodic advertising clock accuracy

struct **esp_ble_dtm_enh_tx_t**

DTM TX parameters.

Public Members

uint8_t **tx_channel**

channel for sending test data, tx_channel = (Frequency -2402)/2, tx_channel range:0x00-0x27, Frequency range: 2402 MHz to 2480 MHz

uint8_t **len_of_data**

length in bytes of payload data in each packet

esp_ble_dtm_pkt_payload_t **pkt_payload**

packet payload type. value range: 0x00-0x07

esp_ble_gap_phy_t **phy**

the phy type used by the transmitter, coded phy with S=2:0x04

struct **esp_ble_dtm_enh_rx_t**

DTM RX parameters.

Public Members

uint8_t **rx_channel**

channel for test data reception, rx_channel = (Frequency -2402)/2, tx_channel range:0x00-0x27, Frequency range: 2402 MHz to 2480 MHz

esp_ble_gap_phy_t **phy**

the phy type used by the receiver, 1M phy: 0x01, 2M phy:0x02, coded phy:0x03

uint8_t **modulation_idx**

modulation index, 0x00:standard modulation index, 0x01:stable modulation index

struct **esp_ble_gap_past_params_t**

periodic adv sync transfer parameters

Public Members

esp_ble_gap_past_mode_t **mode**

periodic advertising sync transfer mode

uint16_t **skip**

the number of periodic advertising packets that can be skipped

uint16_t **sync_timeout**

synchronization timeout for the periodic advertising train

uint8_t **cte_type**

periodic advertising sync transfer CET type

Macros

ESP_BLE_ADV_FLAG_LIMIT_DISC

BLE_ADV_DATA_FLAG data flag bit definition used for advertising data flag.

ESP_BLE_ADV_FLAG_GEN_DISC

ESP_BLE_ADV_FLAG_BREDR_NOT_SPT

ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT

ESP_BLE_ADV_FLAG_DMT_HOST_SPT

ESP_BLE_ADV_FLAG_NON_LIMIT_DISC

ESP_LE_KEY_NONE

relate to BTM_LE_KEY_xxx in stack/btm_api.h

No encryption key

ESP_LE_KEY_PENC

encryption key, encryption information of peer device

ESP_LE_KEY_PID

identity key of the peer device

ESP_LE_KEY_PCSRK

peer SRK

ESP_LE_KEY_PLK

Link key

ESP_LE_KEY_LLK

peer link key

ESP_LE_KEY_LENC

master role security information:div

ESP_LE_KEY_LID

master device ID key

ESP_LE_KEY_LCSRK

local CSRK has been deliver to peer

ESP_LE_AUTH_NO_BOND

relate to BTM_LE_AUTH_xxx in stack/btm_api.h

0 no bondingv

ESP_LE_AUTH_BOND

1 << 0 device in the bonding with peer

ESP_LE_AUTH_REQ_MITM

1 << 2 man in the middle attack

ESP_LE_AUTH_REQ_BOND_MITM

0101 banding with man in the middle attack

ESP_LE_AUTH_REQ_SC_ONLY

1 << 3 secure connection

ESP_LE_AUTH_REQ_SC_BOND

1001 secure connection with band

ESP_LE_AUTH_REQ_SC_MITM

1100 secure conn with MITM

ESP_LE_AUTH_REQ_SC_MITM_BOND

1101 SC with MITM and Bonding

ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE

authentication disable

ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE

authentication enable

ESP_BLE_OOB_DISABLE

disbale the out of bond

ESP_BLE_OOB_ENABLE

enable the out of bond

ESP_IO_CAP_OUT

relate to BTM_IO_CAP_xxx in stack/btm_api.h

DisplayOnly

ESP_IO_CAP_IO

DisplayYesNo

ESP_IO_CAP_IN

KeyboardOnly

ESP_IO_CAP_NONE

NoInputNoOutput

ESP_IO_CAP_KBDISP

Keyboard display

ESP_BLE_APPEARANCE_UNKNOWN

relate to BTM_BLE_APPEARANCE_UNKNOWN in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_PHONE

relate to BTM_BLE_APPEARANCE_GENERIC_PHONE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_COMPUTER

relate to BTM_BLE_APPEARANCE_GENERIC_COMPUTER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_WATCH

relate to BTM_BLE_APPEARANCE_GENERIC_WATCH in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_SPORTS_WATCH

relate to BTM_BLE_APPEARANCE_SPORTS_WATCH in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_CLOCK

relate to BTM_BLE_APPEARANCE_GENERIC_CLOCK in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_DISPLAY

relate to BTM_BLE_APPEARANCE_GENERIC_DISPLAY in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_REMOTE

relate to BTM_BLE_APPEARANCE_GENERIC_REMOTE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES

relate to BTM_BLE_APPEARANCE_GENERIC_EYEGLASSES in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_TAG

relate to BTM_BLE_APPEARANCE_GENERIC_TAG in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_KEYRING

relate to BTM_BLE_APPEARANCE_GENERIC_KEYRING in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER

relate to BTM_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER

relate to BTM_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_THERMOMETER

relate to BTM_BLE_APPEARANCE_GENERIC_THERMOMETER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_THERMOMETER_EAR

relate to BTM_BLE_APPEARANCE_THERMOMETER_EAR in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_HEART_RATE

relate to BTM_BLE_APPEARANCE_GENERIC_HEART_RATE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HEART_RATE_BELT

relate to BTM_BLE_APPEARANCE_HEART_RATE_BELT in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE

relate to BTM_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM

relate to BTM_BLE_APPEARANCE_BLOOD_PRESSURE_ARM in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST

relate to BTM_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_HID

relate to BTM_BLE_APPEARANCE_GENERIC_HID in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_KEYBOARD

relate to BTM_BLE_APPEARANCE_HID_KEYBOARD in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_MOUSE

relate to BTM_BLE_APPEARANCE_HID_MOUSE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_JOYSTICK

relate to BTM_BLE_APPEARANCE_HID_JOYSTICK in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_GAMEPAD

relate to BTM_BLE_APPEARANCE_HID_GAMEPAD in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET

relate to BTM_BLE_APPEARANCE_HID_DIGITIZER_TABLET in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_CARD_READER

relate to BTM_BLE_APPEARANCE_HID_CARD_READER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_DIGITAL_PEN

relate to BTM_BLE_APPEARANCE_HID_DIGITAL_PEN in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER

relate to BTM_BLE_APPEARANCE_HID_BARCODE_SCANNER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_GLUCOSE

relate to BTM_BLE_APPEARANCE_GENERIC_GLUCOSE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_WALKING

relate to BTM_BLE_APPEARANCE_GENERIC_WALKING in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_WALKING_IN_SHOE

relate to BTM_BLE_APPEARANCE_WALKING_IN_SHOE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_WALKING_ON_SHOE

relate to BTM_BLE_APPEARANCE_WALKING_ON_SHOE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_WALKING_ON_HIP

relate to BTM_BLE_APPEARANCE_WALKING_ON_HIP in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_CYCLING

relate to BTM_BLE_APPEARANCE_GENERIC_CYCLING in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_CYCLING_COMPUTER

relate to BTM_BLE_APPEARANCE_CYCLING_COMPUTER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_CYCLING_SPEED

relate to BTM_BLE_APPEARANCE_CYCLING_SPEED in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_CYCLING_CADENCE

relate to BTM_BLE_APPEARANCE_CYCLING_CADENCE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_CYCLING_POWER

relate to BTM_BLE_APPEARANCE_CYCLING_POWER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE

relate to BTM_BLE_APPEARANCE_CYCLING_SPEED_CADENCE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_STANDALONE_SPEAKER

relate to BTM_BLE_APPEARANCE_STANDALONE_SPEAKER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER

relate to BTM_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP

relate to BTM_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST

relate to BTM_BLE_APPEARANCE_PULSE_OXIMETER_WRIST in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_WEIGHT

relate to BTM_BLE_APPEARANCE_GENERIC_WEIGHT in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE

relate to BTM_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR

relate to BTM_BLE_APPEARANCE_POWERED_WHEELCHAIR in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_MOBILITY_SCOOTER

relate to BTM_BLE_APPEARANCE_MOBILITY_SCOOTER in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITOR

relate to BTM_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITOR in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP

relate to BTM_BLE_APPEARANCE_GENERIC_INSULIN_PUMP in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP

relate to BTM_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP

relate to BTM_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_INSULIN_PEN

relate to BTM_BLE_APPEARANCE_INSULIN_PEN in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY

relate to BTM_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS

relate to BTM_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION

relate to BTM_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV

relate to BTM_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD

relate to BTM_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD in stack/btm_ble_api.h

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV

relate to BTM_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV in stack/btm_ble_api.h

BLE_DTM_PKT_PAYLOAD_0x00

PRBS9 sequence '11111111100000111101…' (in transmission order) as described in [Vol 6] Part F, Section 4.1.5

BLE_DTM_PKT_PAYLOAD_0x01

Repeated '11110000' (in transmission order) sequence as described in [Vol 6] Part F, Section 4.1.5

BLE_DTM_PKT_PAYLOAD_0x02

Repeated '10101010' (in transmission order) sequence as described in [Vol 6] Part F, Section 4.1.5

BLE_DTM_PKT_PAYLOAD_0x03

PRBS15 sequence as described in [Vol 6] Part F, Section 4.1.5

BLE_DTM_PKT_PAYLOAD_0x04

Repeated '11111111' (in transmission order) sequence

BLE_DTM_PKT_PAYLOAD_0x05

Repeated '00000000' (in transmission order) sequence

BLE_DTM_PKT_PAYLOAD_0x06

Repeated '00001111' (in transmission order) sequence

BLE_DTM_PKT_PAYLOAD_0x07

Repeated '01010101' (in transmission order) sequence

BLE_DTM_PKT_PAYLOAD_MAX

0x08 ~ 0xFF, Reserved for future use

ESP_GAP_BLE_CHANNELS_LEN

channel length

ESP_GAP_BLE_ADD_WHITELIST_COMPLETE_EVT

This is the old name, just for backwards compatibility.

ESP_BLE_ADV_DATA_LEN_MAX

Advertising data maximum length.

ESP_BLE_SCAN_RSP_DATA_LEN_MAX

Scan response data maximum length.

VENDOR_HCI_CMD_MASK

BLE_BIT(n)

ESP_BLE_GAP_SET_EXT_ADV_PROP_NONCONN_NONSCANNABLE_UNDIRECTED

Non-Connectable and Non-Scannable Undirected advertising

ESP_BLE_GAP_SET_EXT_ADV_PROP_CONNECTABLE

Connectable advertising

ESP_BLE_GAP_SET_EXT_ADV_PROP_SCANNABLE

Scannable advertising

ESP_BLE_GAP_SET_EXT_ADV_PROP_DIRECTED

Directed advertising

ESP_BLE_GAP_SET_EXT_ADV_PROP_HD_DIRECTED

High Duty Cycle Directed Connectable advertising (<= 3.75 ms Advertising Interval)

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY

Use legacy advertising PDUs

ESP_BLE_GAP_SET_EXT_ADV_PROP_ANON_ADV

Omit advertiser’ s address from all PDUs (“anonymous advertising”)

ESP_BLE_GAP_SET_EXT_ADV_PROP_INCLUDE_TX_PWR

Include TxPower in the extended header of the advertising PDU

ESP_BLE_GAP_SET_EXT_ADV_PROP_MASK

Reserved for future use If extended advertising PDU types are being used (bit 4 = 0) then: The advertisement shall not be both connectable and scannable. High duty cycle directed connectable advertising (<= 3.75 ms advertising interval) shall not be used (bit 3 = 0) ADV_IND

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_IND

ADV_DIRECT_IND (low duty cycle)

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_LD_DIR

ADV_DIRECT_IND (high duty cycle)

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_HD_DIR

ADV_SCAN_IND

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_SCAN

ADV_NONCONN_IND

ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_NONCONN

ESP_BLE_GAP_PHY_1M

Secondery Advertisement PHY is LE1M

ESP_BLE_GAP_PHY_2M

Secondery Advertisement PHY is LE2M

ESP_BLE_GAP_PHY_CODED

Secondery Advertisement PHY is LE Coded

ESP_BLE_GAP_NO_PREFER_TRANSMIT_PHY

No Prefer TX PHY supported by controller

ESP_BLE_GAP_NO_PREFER_RECEIVE_PHY

No Prefer RX PHY supported by controller

ESP_BLE_GAP_PRI_PHY_1M

Primary phy only support 1M and LE coded phy.

Primary Phy is LE1M

ESP_BLE_GAP_PRI_PHY_CODED

Primary Phy is LE CODED

ESP_BLE_GAP_PHY_1M_PREF_MASK

The Host prefers use the LE1M transmitter or reciever PHY

ESP_BLE_GAP_PHY_2M_PREF_MASK

The Host prefers use the LE2M transmitter or reciever PHY

ESP_BLE_GAP_PHY_CODED_PREF_MASK

The Host prefers use the LE CODED transmitter or reciever PHY

ESP_BLE_GAP_PHY_OPTIONS_NO_PREF

The Host has no preferred coding when transmitting on the LE Coded PHY

ESP_BLE_GAP_PHY_OPTIONS_PREF_S2_CODING

The Host prefers that S=2 coding be used when transmitting on the LE Coded PHY

ESP_BLE_GAP_PHY_OPTIONS_PREF_S8_CODING

The Host prefers that S=8 coding be used when transmitting on the LE Coded PHY

ESP_BLE_GAP_EXT_SCAN_CFG_UNCODE_MASK

Scan Advertisements on the LE1M PHY

ESP_BLE_GAP_EXT_SCAN_CFG_CODE_MASK

Scan advertisements on the LE coded PHY

ESP_BLE_GAP_EXT_ADV_DATA_COMPLETE

Advertising data.

extended advertising data compete

ESP_BLE_GAP_EXT_ADV_DATA_INCOMPLETE

extended advertising data incomplete

ESP_BLE_GAP_EXT_ADV_DATA_TRUNCATED

extended advertising data truncated mode

ESP_BLE_GAP_SYNC_POLICY_BY_ADV_INFO

Advertising SYNC policy.

sync policy by advertising info

ESP_BLE_GAP_SYNC_POLICY_BY_PERIODIC_LIST

periodic advertising sync policy

ESP_BLE_ADV_REPORT_EXT_ADV_IND

Advertising report.

advertising report with extended advertising indication type

ESP_BLE_ADV_REPORT_EXT_SCAN_IND

advertising report with extended scan indication type

ESP_BLE_ADV_REPORT_EXT_DIRECT_ADV

advertising report with extended direct advertising indication type

ESP_BLE_ADV_REPORT_EXT_SCAN_RSP

advertising report with extended scan response indication type Bluetooth 5.0, Vol 2, Part E, 7.7.65.13

ESP_BLE_LEGACY_ADV_TYPE_IND

advertising report with legacy advertising indication type

ESP_BLE_LEGACY_ADV_TYPE_DIRECT_IND

advertising report with legacy direct indication type

ESP_BLE_LEGACY_ADV_TYPE_SCAN_IND

advertising report with legacy scan indication type

ESP_BLE_LEGACY_ADV_TYPE_NONCON_IND

advertising report with legacy non connectable indication type

ESP_BLE_LEGACY_ADV_TYPE_SCAN_RSP_TO_ADV_IND

advertising report with legacy scan response indication type

ESP_BLE_LEGACY_ADV_TYPE_SCAN_RSP_TO_ADV_SCAN_IND

advertising report with legacy advertising with scan response indication type

EXT_ADV_TX_PWR_NO_PREFERENCE

Extend advertising tx power, range: [-127, +126] dBm.

host has no preference for tx power

ESP_BLE_GAP_PAST_MODE_NO_SYNC_EVT

Periodic advertising sync trans mode.

No attempt is made to sync and no periodic adv sync transfer received event

ESP_BLE_GAP_PAST_MODE_NO_REPORT_EVT

An periodic adv sync transfer received event and no periodic adv report events

ESP_BLE_GAP_PAST_MODE_DUP_FILTER_DISABLED

Periodic adv report events will be enabled with duplicate filtering disabled

ESP_BLE_GAP_PAST_MODE_DUP_FILTER_ENABLED

Periodic adv report events will be enabled with duplicate filtering enabled

Type Definitions

typedef uint8_t **esp_ble_key_type_t**

typedef uint8_t **esp_ble_auth_req_t**

combination of the above bit pattern

typedef uint8_t **esp_ble_io_cap_t**

combination of the io capability

typedef uint8_t **esp_ble_dtm_pkt_payload_t**

typedef uint8_t **esp_gap_ble_channels**[ESP_GAP_BLE_CHANNELS_LEN]

typedef uint8_t **esp_duplicate_info_t**[ESP_BD_ADDR_LEN]

typedef uint16_t **esp_ble_ext_adv_type_mask_t**

typedef uint8_t **esp_ble_gap_phy_t**

typedef uint8_t **esp_ble_gap_all_phys_t**

typedef uint8_t **esp_ble_gap_pri_phy_t**

typedef uint8_t **esp_ble_gap_phy_mask_t**

typedef uint16_t **esp_ble_gap_prefer_phy_options_t**

typedef uint8_t **esp_ble_ext_scan_cfg_mask_t**

typedef uint8_t **esp_ble_gap_ext_adv_data_status_t**

typedef uint8_t **esp_ble_gap_sync_t**

typedef uint8_t **esp_ble_gap_adv_type_t**

typedef uint8_t **esp_ble_gap_past_mode_t**

typedef void (***esp_gap_ble_cb_t**)(*esp_gap_ble_cb_event_t* event, *esp_ble_gap_cb_param_t* *param)

GAP callback function type.

Param event : Event type
Param param : Point to callback parameter, currently is union type

Enumerations

enum **esp_gap_ble_cb_event_t**

GAP BLE callback event type.

Values:

enumerator **ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT**

When advertising data set complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT**

When scan response data set complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT**

When scan parameters set complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_RESULT_EVT**

When one scan result ready, the event comes each time

enumerator **ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT**

When raw advertising data set complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT**

When raw scan response data set complete, the event comes

enumerator **ESP_GAP_BLE_ADV_START_COMPLETE_EVT**

When start advertising complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_START_COMPLETE_EVT**

When start scan complete, the event comes

enumerator **ESP_GAP_BLE_AUTH_CMPL_EVT**

Authentication complete indication.

enumerator **ESP_GAP_BLE_KEY_EVT**

BLE key event for peer device keys

enumerator **ESP_GAP_BLE_SEC_REQ_EVT**

BLE security request

enumerator **ESP_GAP_BLE_PASSKEY_NOTIF_EVT**

passkey notification event

enumerator **ESP_GAP_BLE_PASSKEY_REQ_EVT**

passkey request event

enumerator **ESP_GAP_BLE_OOB_REQ_EVT**

OOB request event

enumerator **ESP_GAP_BLE_LOCAL_IR_EVT**

BLE local IR (identity Root 128-bit random static value used to generate Long Term Key) event

enumerator **ESP_GAP_BLE_LOCAL_ER_EVT**

BLE local ER (Encryption Root vakue used to genrate identity resolving key) event

enumerator **ESP_GAP_BLE_NC_REQ_EVT**

Numeric Comparison request event

enumerator **ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT**

When stop adv complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT**

When stop scan complete, the event comes

enumerator **ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT**

When set the static rand address complete, the event comes

enumerator **ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT**

When update connection parameters complete, the event comes

enumerator **ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT**

When set pkt length complete, the event comes

enumerator **ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT**

When Enable/disable privacy on the local device complete, the event comes

enumerator **ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT**

When remove the bond device complete, the event comes

enumerator **ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT**

When clear the bond device clear complete, the event comes

enumerator **ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT**

When get the bond device list complete, the event comes

enumerator **ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT**

When read the rssi complete, the event comes

enumerator **ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT**

When add or remove whitelist complete, the event comes

enumerator **ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT**

When update duplicate exceptional list complete, the event comes

enumerator **ESP_GAP_BLE_SET_CHANNELS_EVT**

When setting BLE channels complete, the event comes

enumerator **ESP_GAP_BLE_READ_PHY_COMPLETE_EVT**

when reading phy complete, this event comes

enumerator **ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT**

when preferred default phy complete, this event comes

enumerator **ESP_GAP_BLE_SET_PREFERRED_PHY_COMPLETE_EVT**

when preferred phy complete , this event comes

enumerator **ESP_GAP_BLE_EXT_ADV_SET_RAND_ADDR_COMPLETE_EVT**

when extended set random address complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_SET_PARAMS_COMPLETE_EVT**

when extended advertising parameter complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_DATA_SET_COMPLETE_EVT**

when extended advertising data complete, the event comes

enumerator **ESP_GAP_BLE_EXT_SCAN_RSP_DATA_SET_COMPLETE_EVT**

when extended scan response data complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_START_COMPLETE_EVT**

when extended advertising start complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_STOP_COMPLETE_EVT**

when extended advertising stop complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_SET_REMOVE_COMPLETE_EVT**

when extended advertising set remove complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_SET_CLEAR_COMPLETE_EVT**

when extended advertising set clear complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SET_PARAMS_COMPLETE_EVT**

when periodic advertising parameter complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_DATA_SET_COMPLETE_EVT**

when periodic advertising data complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_START_COMPLETE_EVT**
when periodic advertising start complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_STOP_COMPLETE_EVT**
when periodic advertising stop complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_CREATE_SYNC_COMPLETE_EVT**
when periodic advertising create sync complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_CANCEL_COMPLETE_EVT**
when extended advertising sync cancel complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_TERMINATE_COMPLETE_EVT**
when extended advertising sync terminate complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_ADD_DEV_COMPLETE_EVT**
when extended advertising add device complete , the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_REMOVE_DEV_COMPLETE_EVT**
when extended advertising remove device complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_CLEAR_DEV_COMPLETE_EVT**
when extended advertising clear device, the event comes

enumerator **ESP_GAP_BLE_SET_EXT_SCAN_PARAMS_COMPLETE_EVT**
when extended scan parameter complete, the event comes

enumerator **ESP_GAP_BLE_EXT_SCAN_START_COMPLETE_EVT**
when extended scan start complete, the event comes

enumerator **ESP_GAP_BLE_EXT_SCAN_STOP_COMPLETE_EVT**
when extended scan stop complete, the event comes

enumerator **ESP_GAP_BLE_PREFER_EXT_CONN_PARAMS_SET_COMPLETE_EVT**
when extended prefer connection parameter set complete, the event comes

enumerator **ESP_GAP_BLE_PHY_UPDATE_COMPLETE_EVT**
when ble phy update complete, the event comes

enumerator **ESP_GAP_BLE_EXT_ADV_REPORT_EVT**
when extended advertising report complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_TIMEOUT_EVT**
when scan timeout complete, the event comes

enumerator **ESP_GAP_BLE_ADV_TERMINATED_EVT**
when advertising terminate data complete, the event comes

enumerator **ESP_GAP_BLE_SCAN_REQ_RECEIVED_EVT**
when scan req received complete, the event comes

enumerator **ESP_GAP_BLE_CHANNEL_SELECT_ALGORITHM_EVT**
when channel select algorithm complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_REPORT_EVT**
when periodic report advertising complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_LOST_EVT**
when periodic advertising sync lost complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_ESTAB_EVT**
when periodic advertising sync establish complete, the event comes

enumerator **ESP_GAP_BLE_SC_OOB_REQ_EVT**
Secure Connection OOB request event

enumerator **ESP_GAP_BLE_SC_CR_LOC_OOB_EVT**
Secure Connection create OOB data complete event

enumerator **ESP_GAP_BLE_GET_DEV_NAME_COMPLETE_EVT**
When getting BT device name complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_RECV_ENABLE_COMPLETE_EVT**
when set periodic advertising receive enable complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_COMPLETE_EVT**
when periodic advertising sync transfer complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SET_INFO_TRANS_COMPLETE_EVT**
when periodic advertising set info transfer complete, the event comes

enumerator **ESP_GAP_BLE_SET_PAST_PARAMS_COMPLETE_EVT**
when set periodic advertising sync transfer params complete, the event comes

enumerator **ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_RECV_EVT**
when periodic advertising sync transfer received, the event comes

enumerator **ESP_GAP_BLE_DTM_TEST_UPDATE_EVT**
when direct test mode state changes, the event comes

enumerator **ESP_GAP_BLE_ADV_CLEAR_COMPLETE_EVT**
When clear advertising complete, the event comes

enumerator **ESP_GAP_BLE_VENDOR_CMD_COMPLETE_EVT**
When vendor hci command complete, the event comes

enumerator **ESP_GAP_BLE_EVT_MAX**

when maximum advertising event complete, the event comes

enum **esp_ble_adv_data_type**

The type of advertising data(not adv_type)

Values:

enumerator **ESP_BLE_AD_TYPE_FLAG**

enumerator **ESP_BLE_AD_TYPE_16SRV_PART**

enumerator **ESP_BLE_AD_TYPE_16SRV_CMPL**

enumerator **ESP_BLE_AD_TYPE_32SRV_PART**

enumerator **ESP_BLE_AD_TYPE_32SRV_CMPL**

enumerator **ESP_BLE_AD_TYPE_128SRV_PART**

enumerator **ESP_BLE_AD_TYPE_128SRV_CMPL**

enumerator **ESP_BLE_AD_TYPE_NAME_SHORT**

enumerator **ESP_BLE_AD_TYPE_NAME_CMPL**

enumerator **ESP_BLE_AD_TYPE_TX_PWR**

enumerator **ESP_BLE_AD_TYPE_DEV_CLASS**

enumerator **ESP_BLE_AD_TYPE_SM_TK**

enumerator **ESP_BLE_AD_TYPE_SM_OOB_FLAG**

enumerator **ESP_BLE_AD_TYPE_INT_RANGE**

enumerator **ESP_BLE_AD_TYPE_SOL_SRV_UUID**

enumerator **ESP_BLE_AD_TYPE_128SOL_SRV_UUID**

enumerator **ESP_BLE_AD_TYPE_SERVICE_DATA**

enumerator **ESP_BLE_AD_TYPE_PUBLIC_TARGET**

enumerator **ESP_BLE_AD_TYPE_RANDOM_TARGET**

enumerator **ESP_BLE_AD_TYPE_APPEARANCE**

enumerator **ESP_BLE_AD_TYPE_ADV_INT**

enumerator **ESP_BLE_AD_TYPE_LE_DEV_ADDR**

enumerator **ESP_BLE_AD_TYPE_LE_ROLE**

enumerator **ESP_BLE_AD_TYPE_SPAIR_C256**

enumerator **ESP_BLE_AD_TYPE_SPAIR_R256**

enumerator **ESP_BLE_AD_TYPE_32SOL_SRV_UUID**

enumerator **ESP_BLE_AD_TYPE_32SERVICE_DATA**

enumerator **ESP_BLE_AD_TYPE_128SERVICE_DATA**

enumerator **ESP_BLE_AD_TYPE_LE_SECURE_CONFIRM**

enumerator **ESP_BLE_AD_TYPE_LE_SECURE_RANDOM**

enumerator **ESP_BLE_AD_TYPE_URI**

enumerator **ESP_BLE_AD_TYPE_INDOOR_POSITION**

enumerator **ESP_BLE_AD_TYPE_TRANS_DISC_DATA**

enumerator **ESP_BLE_AD_TYPE_LE_SUPPORT_FEATURE**

enumerator **ESP_BLE_AD_TYPE_CHAN_MAP_UPDATE**

enumerator **ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE**

enum **esp_ble_adv_type_t**

Advertising mode.

Values:

enumerator **ADV_TYPE_IND**

enumerator **ADV_TYPE_DIRECT_IND_HIGH**

enumerator **ADV_TYPE_SCAN_IND**

enumerator **ADV_TYPE_NONCONN_IND**

enumerator **ADV_TYPE_DIRECT_IND_LOW**

enum **esp_ble_adv_channel_t**

Advertising channel mask.

Values:

enumerator **ADV_CHNL_37**

enumerator **ADV_CHNL_38**

enumerator **ADV_CHNL_39**

enumerator **ADV_CHNL_ALL**

enum **esp_ble_adv_filter_t**

Values:

enumerator **ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY**

Allow both scan and connection requests from anyone.

enumerator **ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY**

Allow both scan req from White List devices only and connection req from anyone.

enumerator **ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST**

Allow both scan req from anyone and connection req from White List devices only.

enumerator **ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST**

Allow scan and connection requests from White List devices only.

enum **esp_ble_sec_act_t**

Values:

enumerator **ESP_BLE_SEC_ENCRYPT**

relate to BTA_DM_BLE_SEC_ENCRYPT in bta/bta_api.h. If the device has already bonded, the stack will used Long Term Key (LTK) to encrypt with the remote device directly. Else if the device hasn't bonded, the stack will used the default authentication request used the esp_ble_gap_set_security_param function set by the user.

enumerator **ESP_BLE_SEC_ENCRYPT_NO_MITM**

relate to BTA_DM_BLE_SEC_ENCRYPT_NO_MITM in bta/bta_api.h. If the device has been already bonded, the stack will check the LTK (Long Term Key) Whether the authentication request has been met, and if met, use the LTK to encrypt with the remote device directly, else re-pair with the remote device. Else if the device hasn't been bonded, the stack will use NO MITM authentication request in the current link instead of using the authreq in the esp_ble_gap_set_security_param function set by the user.

enumerator **ESP_BLE_SEC_ENCRYPT_MITM**

relate to BTA_DM_BLE_SEC_ENCRYPT_MITM in bta/bta_api.h. If the device has been already bonded, the stack will check the LTK (Long Term Key) whether the authentication request has been met, and if met, use the LTK to encrypt with the remote device directly, else re-pair with the remote device. Else if the device hasn't been bonded, the stack will use MITM authentication request in the current link instead of using the authreq in the esp_ble_gap_set_security_param function set by the user.

enum **esp_ble_sm_param_t**

Values:

enumerator **ESP_BLE_SM_PASSKEY**

Authentication requirements of local device

enumerator **ESP_BLE_SM_AUTHEN_REQ_MODE**

The IO capability of local device

enumerator **ESP_BLE_SM_IOCAP_MODE**

Initiator Key Distribution/Generation

enumerator **ESP_BLE_SM_SET_INIT_KEY**

Responder Key Distribution/Generation

enumerator **ESP_BLE_SM_SET_RSP_KEY**

Maximum Encryption key size to support

enumerator **ESP_BLE_SM_MAX_KEY_SIZE**

Minimum Encryption key size requirement from Peer

enumerator **ESP_BLE_SM_MIN_KEY_SIZE**

Set static Passkey

enumerator **ESP_BLE_SM_SET_STATIC_PASSKEY**

Reset static Passkey

enumerator **ESP_BLE_SM_CLEAR_STATIC_PASSKEY**

Accept only specified SMP Authentication requirement

enumerator **ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH**

Enable/Disable OOB support

enumerator **ESP_BLE_SM_OOB_SUPPORT**

Appl encryption key size

enumerator **ESP_BLE_APP_ENC_KEY_SIZE**

authentication max param

enumerator **ESP_BLE_SM_MAX_PARAM**

enum **esp_ble_dtm_update_evt_t**

Values:

enumerator **DTM_TX_START_EVT**

DTM TX start event.

enumerator **DTM_RX_START_EVT**

DTM RX start event.

enumerator **DTM_TEST_STOP_EVT**

DTM test end event.

enum **esp_ble_scan_type_t**

Ble scan type.

Values:

enumerator **BLE_SCAN_TYPE_PASSIVE**

Passive scan

enumerator **BLE_SCAN_TYPE_ACTIVE**

Active scan

enum **esp_ble_scan_filter_t**

Ble scan filter type.

Values:

enumerator **BLE_SCAN_FILTER_ALLOW_ALL**

Accept all :

i. advertisement packets except directed advertising packets not addressed to this device (default).

enumerator **BLE_SCAN_FILTER_ALLOW_ONLY_WLST**

Accept only :

i. advertisement packets from devices where the advertiser’ s address is in the White list.
ii. Directed advertising packets which are not addressed for this device shall be ignored.

enumerator **BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR**

Accept all :

i. undirected advertisement packets, and
ii. directed advertising packets where the initiator address is a resolvable private address, and
iii. directed advertising packets addressed to this device.

enumerator **BLE_SCAN_FILTER_ALLOW_WLIST_RPA_DIR**

Accept all :

i. advertisement packets from devices where the advertiser’ s address is in the White list, and
ii. directed advertising packets where the initiator address is a resolvable private address, and
iii. directed advertising packets addressed to this device.

enum **esp_ble_scan_duplicate_t**

Ble scan duplicate type.

Values:

enumerator **BLE_SCAN_DUPLICATE_DISABLE**

the Link Layer should generate advertising reports to the host for each packet received

enumerator **BLE_SCAN_DUPLICATE_ENABLE**

the Link Layer should filter out duplicate advertising reports to the Host

enumerator **BLE_SCAN_DUPLICATE_ENABLE_RESET**

Duplicate filtering enabled, reset for each scan period, only supported in BLE 5.0.

enumerator **BLE_SCAN_DUPLICATE_MAX**

Reserved for future use.

enum **esp_gap_search_evt_t**

Sub Event of ESP_GAP_BLE_SCAN_RESULT_EVT.

Values:

enumerator **ESP_GAP_SEARCH_INQ_RES_EVT**

Inquiry result for a peer device.

enumerator **ESP_GAP_SEARCH_INQ_CMPL_EVT**

Inquiry complete.

enumerator **ESP_GAP_SEARCH_DISC_RES_EVT**

Discovery result for a peer device.

enumerator **ESP_GAP_SEARCH_DISC_BLE_RES_EVT**

Discovery result for BLE GATT based service on a peer device.

enumerator **ESP_GAP_SEARCH_DISC_CMPL_EVT**

Discovery complete.

enumerator **ESP_GAP_SEARCH_DI_DISC_CMPL_EVT**

Discovery complete.

enumerator **ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT**

Search cancelled

enumerator **ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT**

The number of pkt discarded by flow control

enum **esp_ble_evt_type_t**

Ble scan result event type, to indicate the result is scan response or advertising data or other.

Values:

enumerator **ESP_BLE_EVT_CONN_ADV**

Connectable undirected advertising (ADV_IND)

enumerator **ESP_BLE_EVT_CONN_DIR_ADV**

Connectable directed advertising (ADV_DIRECT_IND)

enumerator **ESP_BLE_EVT_DISC_ADV**

Scannable undirected advertising (ADV_SCAN_IND)

enumerator **ESP_BLE_EVT_NON_CONN_ADV**

Non connectable undirected advertising (ADV_NONCONN_IND)

enumerator **ESP_BLE_EVT_SCAN_RSP**

Scan Response (SCAN_RSP)

enum **esp_ble_wl_operation_t**

Values:

enumerator **ESP_BLE_WHITELIST_REMOVE**

remove mac from whitelist

enumerator **ESP_BLE_WHITELIST_ADD**

add address to whitelist

enumerator **ESP_BLE_WHITELIST_CLEAR**

clear all device in whitelist

enum **esp_bt_duplicate_exceptional_subcode_type_t**

Values:

enumerator **ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_ADD**

Add device info into duplicate scan exceptional list

enumerator **ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_REMOVE**

Remove device info from duplicate scan exceptional list

enumerator **ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_CLEAN**

Clean duplicate scan exceptional list

enum **esp_ble_duplicate_exceptional_info_type_t**

Values:

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_ADV_ADDR**

BLE advertising address , device info will be added into ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ADDR_LIST

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_LINK_ID**

BLE mesh link ID, it is for BLE mesh, device info will be added into ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_LINK_ID_LIST

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_BEACON_TYPE**

BLE mesh beacon AD type, the format is | Len | 0x2B | Beacon Type | Beacon Data |

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_PROV_SRV_ADV**

BLE mesh provisioning service uuid, the format is | 0x02 | 0x01 | flags | 0x03 | 0x03 | 0x1827 | ···. |`

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_PROXY_SRV_ADV**

BLE mesh adv with proxy service uuid, the format is | 0x02 | 0x01 | flags | 0x03 | 0x03 | 0x1828 | ···. |`

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_PROXY_SOLIC_ADV**
BLE mesh adv with proxy service uuid, the format is | 0x02 | 0x01 | flags | 0x03 | 0x03 | 0x1859 | ···. |`

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_URI_ADV**
BLE mesh URI adv, the format is ···| Len | 0x24 | data |···

enum **esp_duplicate_scan_exceptional_list_type_t**
Values:

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ADDR_LIST**
duplicate scan exceptional addr list

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_LINK_ID_LIST**
duplicate scan exceptional mesh link ID list

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_BEACON_TYPE_LIST**
duplicate scan exceptional mesh beacon type list

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_PROV_SRV_ADV_LIST**
duplicate scan exceptional mesh adv with provisioning service uuid

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_PROXY_SRV_ADV_LIST**
duplicate scan exceptional mesh adv with proxy service uuid

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_PROXY_SOLIC_ADV_LIST**
duplicate scan exceptional mesh adv with proxy solicitation PDU uuid

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_URI_ADV_LIST**
duplicate scan exceptional URI list

enumerator **ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ALL_LIST**
duplicate scan exceptional all list

GATT DEFINES

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_gatt_defs.h

Unions

union **esp_gatt_rsp_t**
#include <esp_gatt_defs.h> GATT remote read request response type.

Public Members

esp_gatt_value_t **attr_value**

Gatt attribute structure

uint16_t **handle**

Gatt attribute handle

Structures

struct **esp_gatt_id_t**

Gatt id, include uuid and instance id.

Public Members

esp_bt_uuid_t **uuid**

UUID

uint8_t **inst_id**

Instance id

struct **esp_gatt_srvc_id_t**

Gatt service id, include id (uuid and instance id) and primary flag.

Public Members

esp_gatt_id_t **id**

Gatt id, include uuid and instance

bool **is_primary**

This service is primary or not

struct **esp_attr_desc_t**

Attribute description (used to create database)

Public Members

uint16_t **uuid_length**

UUID length

uint8_t ***uuid_p**

UUID value

uint16_t **perm**

Attribute permission

uint16_t **max_length**

Maximum length of the element

uint16_t **length**

Current length of the element

uint8_t ***value**

Element value array

struct **esp_attr_control_t**

attribute auto response flag

Public Members

uint8_t **auto_rsp**

if auto_rsp set to ESP_GATT_RSP_BY_APP, means the response of Write/Read operation will by replied by application. if auto_rsp set to ESP_GATT_AUTO_RSP, means the response of Write/Read operation will be replied by GATT stack automatically.

struct **esp_gatts_attr_db_t**

attribute type added to the gatt server database

Public Members

esp_attr_control_t **attr_control**

The attribute control type

esp_attr_desc_t **att_desc**

The attribute type

struct **esp_attr_value_t**

set the attribute value type

Public Members

uint16_t **attr_max_len**

attribute max value length

uint16_t **attr_len**

attribute current value length

uint8_t ***attr_value**

the pointer to attribute value

struct **esp_gatts_incl_svc_desc_t**

Gatt include service entry element.

Public Members

uint16_t **start_hdl**

Gatt start handle value of included service

uint16_t **end_hdl**

Gatt end handle value of included service

uint16_t **uuid**

Gatt attribute value UUID of included service

struct **esp_gatts_incl128_svc_desc_t**

Gatt include 128 bit service entry element.

Public Members

uint16_t **start_hdl**

Gatt start handle value of included 128 bit service

uint16_t **end_hdl**

Gatt end handle value of included 128 bit service

struct **esp_gatt_value_t**

Gatt attribute value.

Public Members

uint8_t **value**[ESP_GATT_MAX_ATTR_LEN]

Gatt attribute value

uint16_t **handle**

Gatt attribute handle

uint16_t **offset**

Gatt attribute value offset

uint16_t **len**

Gatt attribute value length

uint8_t **auth_req**

Gatt authentication request

struct **esp_gatt_conn_params_t**

Connection parameters information.

Public Members

uint16_t **interval**

connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct **esp_gattc_multi_t**

read multiple attribute

Public Members

uint8_t **num_attr**

The number of the attribute

uint16_t **handles**[ESP_GATT_MAX_READ_MULTI_HANDLES]

The handles list

struct **esp_gattc_db_elem_t**

data base attribute element

Public Members

esp_gatt_db_attr_type_t **type**

The attribute type

uint16_t **attribute_handle**

The attribute handle, it's valid for all of the type

uint16_t **start_handle**

The service start handle, it's valid only when the type = ESP_GATT_DB_PRIMARY_SERVICE or ESP_GATT_DB_SECONDARY_SERVICE

uint16_t **end_handle**

The service end handle, it's valid only when the type = ESP_GATT_DB_PRIMARY_SERVICE or ESP_GATT_DB_SECONDARY_SERVICE

esp_gatt_char_prop_t **properties**

The characteristic properties, it's valid only when the type = ESP_GATT_DB_CHARACTERISTIC

esp_bt_uuid_t **uuid**

The attribute uuid, it's valid for all of the type

struct **esp_gattc_service_elem_t**

service element

Public Members

bool **is_primary**

The service flag, true if the service is primary service, else is secondary service

uint16_t **start_handle**

The start handle of the service

uint16_t **end_handle**

The end handle of the service

esp_bt_uuid_t **uuid**

The uuid of the service

struct **esp_gattc_char_elem_t**

characteristic element

Public Members

uint16_t **char_handle**

The characteristic handle

esp_gatt_char_prop_t **properties**

The characteristic properties

esp_bt_uuid_t **uuid**

The characteristic uuid

struct **esp_gattc_descr_elem_t**

descriptor element

Public Members

uint16_t **handle**

The characteristic descriptor handle

esp_bt_uuid_t **uuid**

The characteristic descriptor uuid

struct **esp_gattc_incl_svc_elem_t**

include service element

Public Members

uint16_t **handle**

The include service current attribute handle

uint16_t **incl_srvc_s_handle**

The start handle of the service which has been included

uint16_t **incl_srvc_e_handle**

The end handle of the service which has been included

esp_bt_uuid_t **uuid**

The include service uuid

Macros

ESP_GATT_UUID_IMMEDIATE_ALERT_SVC

All "ESP_GATT_UUID_xxx" is attribute types

ESP_GATT_UUID_LINK_LOSS_SVC

ESP_GATT_UUID_TX_POWER_SVC

ESP_GATT_UUID_CURRENT_TIME_SVC

ESP_GATT_UUID_REF_TIME_UPDATE_SVC

ESP_GATT_UUID_NEXT_DST_CHANGE_SVC

ESP_GATT_UUID_GLUCOSE_SVC

ESP_GATT_UUID_HEALTH_THERMOM_SVC

ESP_GATT_UUID_DEVICE_INFO_SVC

ESP_GATT_UUID_HEART_RATE_SVC

ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC

ESP_GATT_UUID_BATTERY_SERVICE_SVC

ESP_GATT_UUID_BLOOD_PRESSURE_SVC

ESP_GATT_UUID_ALERT_NTF_SVC

ESP_GATT_UUID_HID_SVC

ESP_GATT_UUID_SCAN_PARAMETERS_SVC

ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC

ESP_GATT_UUID_Automation_IO_SVC

ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC

ESP_GATT_UUID_CYCLING_POWER_SVC

ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC

ESP_GATT_UUID_ENVIRONMENTAL_SENSING_SVC

ESP_GATT_UUID_BODY_COMPOSITION

ESP_GATT_UUID_USER_DATA_SVC

ESP_GATT_UUID_WEIGHT_SCALE_SVC

ESP_GATT_UUID_BOND_MANAGEMENT_SVC

ESP_GATT_UUID_CONT_GLUCOSE_MONITOR_SVC

ESP_GATT_UUID_PRI_SERVICE

ESP_GATT_UUID_SEC_SERVICE

ESP_GATT_UUID_INCLUDE_SERVICE

ESP_GATT_UUID_CHAR_DECLARE

ESP_GATT_UUID_CHAR_EXT_PROP

ESP_GATT_UUID_CHAR_DESCRIPTION

ESP_GATT_UUID_CHAR_CLIENT_CONFIG

ESP_GATT_UUID_CHAR_SRVR_CONFIG

ESP_GATT_UUID_CHAR_PRESENT_FORMAT

ESP_GATT_UUID_CHAR_AGG_FORMAT

ESP_GATT_UUID_CHAR_VALID_RANGE

ESP_GATT_UUID_EXT_RPT_REF_DESCR

ESP_GATT_UUID_RPT_REF_DESCR

ESP_GATT_UUID_NUM_DIGITALS_DESCR

ESP_GATT_UUID_VALUE_TRIGGER_DESCR

ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR

ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR

ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR

ESP_GATT_UUID_TIME_TRIGGER_DESCR

ESP_GATT_UUID_GAP_DEVICE_NAME

ESP_GATT_UUID_GAP_ICON

ESP_GATT_UUID_GAP_PREF_CONN_PARAM

ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL

ESP_GATT_UUID_GATT_SRV_CHGD

ESP_GATT_UUID_ALERT_LEVEL

ESP_GATT_UUID_TX_POWER_LEVEL

ESP_GATT_UUID_CURRENT_TIME

ESP_GATT_UUID_LOCAL_TIME_INFO

ESP_GATT_UUID_REF_TIME_INFO

ESP_GATT_UUID_NW_STATUS

ESP_GATT_UUID_NW_TRIGGER

ESP_GATT_UUID_ALERT_STATUS

ESP_GATT_UUID_RINGER_CP

ESP_GATT_UUID_RINGER_SETTING

ESP_GATT_UUID_GM_MEASUREMENT

ESP_GATT_UUID_GM_CONTEXT

ESP_GATT_UUID_GM_CONTROL_POINT

ESP_GATT_UUID_GM_FEATURE

ESP_GATT_UUID_SYSTEM_ID

ESP_GATT_UUID_MODEL_NUMBER_STR

ESP_GATT_UUID_SERIAL_NUMBER_STR

ESP_GATT_UUID_FW_VERSION_STR

ESP_GATT_UUID_HW_VERSION_STR

ESP_GATT_UUID_SW_VERSION_STR

ESP_GATT_UUID_MANU_NAME

ESP_GATT_UUID_IEEE_DATA

ESP_GATT_UUID_PNP_ID

ESP_GATT_UUID_HID_INFORMATION

ESP_GATT_UUID_HID_REPORT_MAP

ESP_GATT_UUID_HID_CONTROL_POINT

ESP_GATT_UUID_HID_REPORT

ESP_GATT_UUID_HID_PROTO_MODE

ESP_GATT_UUID_HID_BT_KB_INPUT

ESP_GATT_UUID_HID_BT_KB_OUTPUT

ESP_GATT_UUID_HID_BT_MOUSE_INPUT

ESP_GATT_HEART_RATE_MEAS
Heart Rate Measurement.

ESP_GATT_BODY_SENSOR_LOCATION
Body Sensor Location.

ESP_GATT_HEART_RATE_CNTL_POINT
Heart Rate Control Point.

ESP_GATT_UUID_BATTERY_LEVEL

ESP_GATT_UUID_SC_CONTROL_POINT

ESP_GATT_UUID_SENSOR_LOCATION

ESP_GATT_UUID_RSC_MEASUREMENT

ESP_GATT_UUID_RSC_FEATURE

ESP_GATT_UUID_CSC_MEASUREMENT

ESP_GATT_UUID_CSC_FEATURE

ESP_GATT_UUID_SCAN_INT_WINDOW

ESP_GATT_UUID_SCAN_REFRESH

ESP_GATT_ILLEGAL_UUID
GATT INVALID UUID.

ESP_GATT_ILLEGAL_HANDLE
GATT INVALID HANDLE.

ESP_GATT_ATTR_HANDLE_MAX
GATT attribute max handle.

ESP_GATT_MAX_READ_MULTI_HANDLES

ESP_GATT_PERM_READ
Attribute permissions.

ESP_GATT_PERM_READ_ENCRYPTED

ESP_GATT_PERM_READ_ENC_MITM

ESP_GATT_PERM_WRITE

ESP_GATT_PERM_WRITE_ENCRYPTED

ESP_GATT_PERM_WRITE_ENC_MITM

ESP_GATT_PERM_WRITE_SIGNED

ESP_GATT_PERM_WRITE_SIGNED_MITM

ESP_GATT_PERM_READ_AUTHORIZATION

ESP_GATT_PERM_WRITE_AUTHORIZATION

ESP_GATT_PERM_ENCRYPT_KEY_SIZE(keysize)

ESP_GATT_CHAR_PROP_BIT_BROADCAST

ESP_GATT_CHAR_PROP_BIT_READ

ESP_GATT_CHAR_PROP_BIT_WRITE_NR

ESP_GATT_CHAR_PROP_BIT_WRITE

ESP_GATT_CHAR_PROP_BIT_NOTIFY

ESP_GATT_CHAR_PROP_BIT_INDICATE

ESP_GATT_CHAR_PROP_BIT_AUTH

ESP_GATT_CHAR_PROP_BIT_EXT_PROP

ESP_GATT_MAX_ATTR_LEN

GATT maximum attribute length.

ESP_GATT_RSP_BY_APP

ESP_GATT_AUTO_RSP

ESP_GATT_IF_NONE

If callback report gattc_if/gatts_if as this macro, means this event is not correspond to any app

Type Definitions

typedef uint16_t **esp_gatt_perm_t**

typedef uint8_t **esp_gatt_char_prop_t**

typedef uint8_t **esp_gatt_if_t**

Gatt interface type, different application on GATT client use different gatt_if

Enumerations

enum **esp_gatt_prep_write_type**

Attribute write data type from the client.

Values:

enumerator **ESP_GATT_PREP_WRITE_CANCEL**

Prepare write cancel

enumerator **ESP_GATT_PREP_WRITE_EXEC**

Prepare write execute

enum **esp_gatt_status_t**

GATT success code and error codes.

Values:

enumerator **ESP_GATT_OK**

enumerator **ESP_GATT_INVALID_HANDLE**

enumerator **ESP_GATT_READ_NOT_PERMIT**

enumerator **ESP_GATT_WRITE_NOT_PERMIT**

enumerator **ESP_GATT_INVALID_PDU**

enumerator **ESP_GATT_INSUF_AUTHENTICATION**

enumerator **ESP_GATT_REQ_NOT_SUPPORTED**

enumerator **ESP_GATT_INVALID_OFFSET**

enumerator **ESP_GATT_INSUF_AUTHORIZATION**

enumerator **ESP_GATT_PREPARE_Q_FULL**

enumerator **ESP_GATT_NOT_FOUND**

enumerator **ESP_GATT_NOT_LONG**

enumerator **ESP_GATT_INSUF_KEY_SIZE**

enumerator **ESP_GATT_INVALID_ATTR_LEN**

enumerator **ESP_GATT_ERR_UNLIKELY**

enumerator **ESP_GATT_INSUF_ENCRYPTION**

enumerator **ESP_GATT_UNSUPPORT_GRP_TYPE**

enumerator **ESP_GATT_INSUF_RESOURCE**

enumerator **ESP_GATT_NO_RESOURCES**

enumerator **ESP_GATT_INTERNAL_ERROR**

enumerator **ESP_GATT_WRONG_STATE**

enumerator **ESP_GATT_DB_FULL**

enumerator **ESP_GATT_BUSY**

enumerator **ESP_GATT_ERROR**

enumerator **ESP_GATT_CMD_STARTED**

enumerator **ESP_GATT_ILLEGAL_PARAMETER**

enumerator **ESP_GATT_PENDING**

enumerator **ESP_GATT_AUTH_FAIL**

enumerator **ESP_GATT_MORE**

enumerator **ESP_GATT_INVALID_CFG**

enumerator **ESP_GATT_SERVICE_STARTED**

enumerator **ESP_GATT_ENCRYPTED_MITM**

enumerator **ESP_GATT_ENCRYPTED_NO_MITM**

enumerator **ESP_GATT_NOT_ENCRYPTED**

enumerator **ESP_GATT_CONGESTED**

enumerator **ESP_GATT_DUP_REG**

enumerator **ESP_GATT_ALREADY_OPEN**

enumerator **ESP_GATT_CANCEL**

enumerator **ESP_GATT_STACK_RSP**

enumerator **ESP_GATT_APP_RSP**

enumerator **ESP_GATT_UNKNOWN_ERROR**

enumerator **ESP_GATT_CCC_CFG_ERR**

enumerator **ESP_GATT_PRC_IN_PROGRESS**

enumerator **ESP_GATT_OUT_OF_RANGE**

enum **esp_gatt_conn_reason_t**

Gatt Connection reason enum.

Values:

enumerator **ESP_GATT_CONN_UNKNOWN**

Gatt connection unknown

enumerator **ESP_GATT_CONN_L2C_FAILURE**

General L2cap failure

enumerator **ESP_GATT_CONN_TIMEOUT**

Connection timeout

enumerator **ESP_GATT_CONN_TERMINATE_PEER_USER**

Connection terminate by peer user

enumerator **ESP_GATT_CONN_TERMINATE_LOCAL_HOST**

Connection terminated by local host

enumerator **ESP_GATT_CONN_FAIL_ESTABLISH**

Connection fail to establish

enumerator **ESP_GATT_CONN_LMP_TIMEOUT**

Connection fail for LMP response tout

enumerator **ESP_GATT_CONN_CONN_CANCEL**

L2CAP connection cancelled

enumerator **ESP_GATT_CONN_NONE**

No connection to cancel

enum **esp_gatt_auth_req_t**

Gatt authentication request type.

Values:

enumerator **ESP_GATT_AUTH_REQ_NONE**

enumerator **ESP_GATT_AUTH_REQ_NO_MITM**

enumerator **ESP_GATT_AUTH_REQ_MITM**

enumerator **ESP_GATT_AUTH_REQ_SIGNED_NO_MITM**

enumerator **ESP_GATT_AUTH_REQ_SIGNED_MITM**

enum **esp_service_source_t**

Values:

enumerator **ESP_GATT_SERVICE_FROM_REMOTE_DEVICE**

enumerator **ESP_GATT_SERVICE_FROM_NVS_FLASH**

enumerator **ESP_GATT_SERVICE_FROM_UNKNOWN**

enum **esp_gatt_write_type_t**

Gatt write type.

Values:

enumerator **ESP_GATT_WRITE_TYPE_NO_RSP**

Gatt write attribute need no response

enumerator **ESP_GATT_WRITE_TYPE_RSP**

Gatt write attribute need remote response

enum **esp_gatt_db_attr_type_t**

the type of attribute element

Values:

enumerator **ESP_GATT_DB_PRIMARY_SERVICE**

Gattc primary service attribute type in the cache

enumerator **ESP_GATT_DB_SECONDARY_SERVICE**

Gattc secondary service attribute type in the cache

enumerator **ESP_GATT_DB_CHARACTERISTIC**

Gattc characteristic attribute type in the cache

enumerator **ESP_GATT_DB_DESCRIPTOR**

Gattc characteristic descriptor attribute type in the cache

enumerator **ESP_GATT_DB_INCLUDED_SERVICE**

Gattc include service attribute type in the cache

enumerator **ESP_GATT_DB_ALL**

Gattc all the attribute (primary service & secondary service & include service & char & descriptor) type in the cache

GATT SERVER API

Application Example Check bluetooth/bluedroid/ble folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a GATT sever demo and its tutorial. This demo creates a GATT service with an attribute table, which releases the user from adding attributes one by one. This is the recommended method of adding attributes.
  - bluetooth/bluedroid/ble/gatt_server_service_table
  - GATT Server Service Table Example Walkthrough
- This is a GATT server demo and its tutorial. This demo creates a GATT service by adding attributes one by one as defined by Bluedroid. The recommended method of adding attributes is presented in example above.
  - bluetooth/bluedroid/ble/gatt_server
  - GATT Server Example Walkthrough
- This is a BLE SPP-Like demo. This demo, which acts as a GATT server, can receive data from UART and then send the data to the peer device automatically.
  - bluetooth/bluedroid/ble/ble_spp_server

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_gatts_api.h

Functions

esp_err_t **esp_ble_gatts_register_callback** (*esp_gatts_cb_t* callback)

This function is called to register application callbacks with BTA GATTS module.

Returns
- ESP_OK : success
- other : failed

esp_gatts_cb_t **esp_ble_gatts_get_callback** (void)

This function is called to get the current application callbacks with BTA GATTS module.

Returns
- esp_gatts_cb_t : current callback

esp_err_t **esp_ble_gatts_app_register** (uint16_t app_id)

This function is called to register application identifier.

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_app_unregister** (*esp_gatt_if_t* gatts_if)

unregister with GATT Server.

Parameters **gatts_if** – **[in]** GATT server access interface

Returns
- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_create_service**(*esp_gatt_if_t* gatts_if, *esp_gatt_srvc_id_t* *service_id, uint16_t num_handle)

Create a service. When service creation is done, a callback event ESP_GATTS_CREATE_EVT is called to report status and service ID to the profile. The service ID obtained in the callback function needs to be used when adding included service and characteristics/descriptors into the service.

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **service_id** –**[in]** service ID.
- **num_handle** –**[in]** number of handle requested for this service.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_create_attr_tab**(const *esp_gatts_attr_db_t* *gatts_attr_db, *esp_gatt_if_t* gatts_if, uint16_t max_nb_attr, uint8_t srvc_inst_id)

Create a service attribute tab.

Parameters

- **gatts_attr_db** –**[in]** the pointer to the service attr tab
- **gatts_if** –**[in]** GATT server access interface
- **max_nb_attr** –**[in]** the number of attribute to be added to the service database.
- **srvc_inst_id** –**[in]** the instance id of the service

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_add_included_service**(uint16_t service_handle, uint16_t included_service_handle)

This function is called to add an included service. This function have to be called between 'esp_ble_gatts_create_service' and 'esp_ble_gatts_add_char' . After included service is included, a callback event ESP_GATTS_ADD_INCL_SRVC_EVT is reported the included service ID.

Parameters

- **service_handle** –**[in]** service handle to which this included service is to be added.
- **included_service_handle** –**[in]** the service ID to be included.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_add_char**(uint16_t service_handle, *esp_bt_uuid_t* *char_uuid, *esp_gatt_perm_t* perm, *esp_gatt_char_prop_t* property, *esp_attr_value_t* *char_val, *esp_attr_control_t* *control)

This function is called to add a characteristic into a service.

Parameters

- **service_handle** –**[in]** service handle to which this included service is to be added.
- **char_uuid** –**[in]** : Characteristic UUID.
- **perm** –**[in]** : Characteristic value declaration attribute permission.
- **property** –**[in]** : Characteristic Properties
- **char_val** –**[in]** : Characteristic value
- **control** –**[in]** : attribute response control byte

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_add_char_descr**(uint16_t service_handle, *esp_bt_uuid_t* *descr_uuid, *esp_gatt_perm_t* perm, *esp_attr_value_t* *char_descr_val, *esp_attr_control_t* *control)

This function is called to add characteristic descriptor. When it' s done, a callback event ESP_GATTS_ADD_DESCR_EVT is called to report the status and an ID number for this descriptor.

Parameters

- **service_handle** –**[in]** service handle to which this characteristic descriptor is to be added.
- **perm** –**[in]** descriptor access permission.
- **descr_uuid** –**[in]** descriptor UUID.
- **char_descr_val** –**[in]** : Characteristic descriptor value
- **control** –**[in]** : attribute response control byte

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_delete_service**(uint16_t service_handle)

This function is called to delete a service. When this is done, a callback event ESP_GATTS_DELETE_EVT is report with the status.

Parameters **service_handle** –**[in]** service_handle to be deleted.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_start_service**(uint16_t service_handle)

This function is called to start a service.

Parameters **service_handle** –**[in]** the service handle to be started.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_stop_service**(uint16_t service_handle)

This function is called to stop a service.

Parameters **service_handle** –**[in]** - service to be topped.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_send_indicate**(*esp_gatt_if_t* gatts_if, uint16_t conn_id, uint16_t attr_handle, uint16_t value_len, uint8_t *value, bool need_confirm)

Send indicate or notify to GATT client. Set param need_confirm as false will send notification, otherwise indication. Note: the size of indicate or notify data need less than MTU size,see "esp_ble_gattc_send_mtu_req" .

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **conn_id** –**[in]** - connection id to indicate.
- **attr_handle** –**[in]** - attribute handle to indicate.
- **value_len** –**[in]** - indicate value length.
- **value** –**[in]** value to indicate.
- **need_confirm** –**[in]** - Whether a confirmation is required. false sends a GATT notification, true sends a GATT indication.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_send_response**(*esp_gatt_if_t* gatts_if, uint16_t conn_id, uint32_t trans_id, *esp_gatt_status_t* status, *esp_gatt_rsp_t* *rsp)

This function is called to send a response to a request.

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **conn_id** –**[in]** - connection identifier.
- **trans_id** –**[in]** - transfer id

- **status** –**[in]** - response status
- **rsp** –**[in]** - response data.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_set_attr_value**(uint16_t attr_handle, uint16_t length, const uint8_t *value)

This function is called to set the attribute value by the application.

Parameters

- **attr_handle** –**[in]** the attribute handle which to be set
- **length** –**[in]** the value length
- **value** –**[in]** the pointer to the attribute value

Returns

- ESP_OK : success
- other : failed

esp_gatt_status_t **esp_ble_gatts_get_attr_value**(uint16_t attr_handle, uint16_t *length, const uint8_t **value)

Retrieve attribute value.

Parameters

- **attr_handle** –**[in]** Attribute handle.
- **length** –**[out]** pointer to the attribute value length
- **value** –**[out]** Pointer to attribute value payload, the value cannot be modified by user

Returns

- ESP_GATT_OK : success
- other : failed

esp_err_t **esp_ble_gatts_open**(*esp_gatt_if_t* gatts_if, *esp_bd_addr_t* remote_bda, bool is_direct)

Open a direct open connection or add a background auto connection.

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **remote_bda** –**[in]** remote device bluetooth device address.
- **is_direct** –**[in]** direct connection or background auto connection

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_close**(*esp_gatt_if_t* gatts_if, uint16_t conn_id)

Close a connection a remote device.

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **conn_id** –**[in]** connection ID to be closed.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_send_service_change_indication**(*esp_gatt_if_t* gatts_if, *esp_bd_addr_t* remote_bda)

Send service change indication.

Parameters

- **gatts_if** –**[in]** GATT server access interface
- **remote_bda** –**[in]** remote device bluetooth device address. If remote_bda is NULL then it will send service change indication to all the connected devices and if not then to a specific device

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_show_local_database**(void)

Print local database (GATT service table)

Returns

- ESP_OK : success
- other : failed

Unions

union **esp_ble_gatts_cb_param_t**

#include <esp_gatts_api.h> Gatt server callback parameters union.

Public Members

struct *esp_ble_gatts_cb_param_t*::*gatts_reg_evt_param* **reg**

Gatt server callback param of ESP_GATTS_REG_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_read_evt_param* **read**

Gatt server callback param of ESP_GATTS_READ_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_write_evt_param* **write**

Gatt server callback param of ESP_GATTS_WRITE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_exec_write_evt_param* **exec_write**

Gatt server callback param of ESP_GATTS_EXEC_WRITE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_mtu_evt_param* **mtu**

Gatt server callback param of ESP_GATTS_MTU_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_conf_evt_param* **conf**

Gatt server callback param of ESP_GATTS_CONF_EVT (confirm)

struct *esp_ble_gatts_cb_param_t*::*gatts_create_evt_param* **create**

Gatt server callback param of ESP_GATTS_CREATE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_add_incl_srvc_evt_param* **add_incl_srvc**

Gatt server callback param of ESP_GATTS_ADD_INCL_SRVC_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_add_char_evt_param* **add_char**

Gatt server callback param of ESP_GATTS_ADD_CHAR_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_add_char_descr_evt_param* **add_char_descr**

Gatt server callback param of ESP_GATTS_ADD_CHAR_DESCR_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_delete_evt_param* **del**

Gatt server callback param of ESP_GATTS_DELETE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_start_evt_param* **start**

Gatt server callback param of ESP_GATTS_START_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_stop_evt_param* **stop**

Gatt server callback param of ESP_GATTS_STOP_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_connect_evt_param* **connect**

Gatt server callback param of ESP_GATTS_CONNECT_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_disconnect_evt_param* **disconnect**

Gatt server callback param of ESP_GATTS_DISCONNECT_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_open_evt_param* **open**

Gatt server callback param of ESP_GATTS_OPEN_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_cancel_open_evt_param* **cancel_open**

Gatt server callback param of ESP_GATTS_CANCEL_OPEN_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_close_evt_param* **close**

Gatt server callback param of ESP_GATTS_CLOSE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_congest_evt_param* **congest**

Gatt server callback param of ESP_GATTS_CONGEST_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_rsp_evt_param* **rsp**

Gatt server callback param of ESP_GATTS_RESPONSE_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_add_attr_tab_evt_param* **add_attr_tab**

Gatt server callback param of ESP_GATTS_CREAT_ATTR_TAB_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_set_attr_val_evt_param* **set_attr_val**

Gatt server callback param of ESP_GATTS_SET_ATTR_VAL_EVT

struct *esp_ble_gatts_cb_param_t*::*gatts_send_service_change_evt_param* **service_change**

Gatt server callback param of ESP_GATTS_SEND_SERVICE_CHANGE_EVT

struct **gatts_add_attr_tab_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_CREAT_ATTR_TAB_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

esp_bt_uuid_t **svc_uuid**

Service uuid type

uint8_t **svc_inst_id**

Service id

uint16_t **num_handle**

The number of the attribute handle to be added to the gatts database

uint16_t ***handles**

The number to the handles

struct **gatts_add_char_descr_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_ADD_CHAR_DESCR_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **attr_handle**

Descriptor attribute handle

uint16_t **service_handle**

Service attribute handle

esp_bt_uuid_t **descr_uuid**

Characteristic descriptor uuid

struct **gatts_add_char_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_ADD_CHAR_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **attr_handle**

Characteristic attribute handle

uint16_t **service_handle**

Service attribute handle

esp_bt_uuid_t **char_uuid**

Characteristic uuid

struct **gatts_add_incl_srvc_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_ADD_INCL_SRVC_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **attr_handle**

Included service attribute handle

uint16_t **service_handle**

Service attribute handle

struct **gatts_cancel_open_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_CANCEL_OPEN_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

struct **gatts_close_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_CLOSE_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

struct **gatts_conf_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_CONF_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

uint16_t **handle**

attribute handle

uint16_t **len**

The indication or notification value length, len is valid when send notification or indication failed

uint8_t ***value**

The indication or notification value , value is valid when send notification or indication failed

struct **gatts_congest_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_LISTEN_EVT.

ESP_GATTS_CONGEST_EVT

Public Members

uint16_t **conn_id**

Connection id

bool **congested**

Congested or not

struct **gatts_connect_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_CONNECT_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint8_t **link_role**

Link role : master role = 0 ; slave role = 1

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

esp_gatt_conn_params_t **conn_params**

current Connection parameters

esp_ble_addr_type_t **ble_addr_type**

Remote BLE device address type

uint16_t **conn_handle**

HCI connection handle

struct **gatts_create_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_UNREG_EVT.

ESP_GATTS_CREATE_EVT

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

esp_gatt_srvc_id_t **service_id**

Service id, include service uuid and other information

struct **gatts_delete_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_DELETE_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

struct **gatts_disconnect_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_DISCONNECT_EVT.

Public Members

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

esp_gatt_conn_reason_t **reason**

Indicate the reason of disconnection

struct **gatts_exec_write_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_EXEC_WRITE_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint32_t **trans_id**

Transfer id

esp_bd_addr_t **bda**

The bluetooth device address which been written

uint8_t **exec_write_flag**

Execute write flag

struct **gatts_mtu_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_MTU_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint16_t **mtu**

MTU size

struct **gatts_open_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_OPEN_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

struct **gatts_read_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_READ_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint32_t **trans_id**

Transfer id

esp_bd_addr_t **bda**

The bluetooth device address which been read

uint16_t **handle**

The attribute handle

uint16_t **offset**

Offset of the value, if the value is too long

bool **is_long**

The value is too long or not

bool **need_rsp**

The read operation need to do response

struct **gatts_reg_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_REG_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **app_id**

Application id which input in register API

struct **gatts_rsp_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_RESPONSE_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **handle**

Attribute handle which send response

struct **gatts_send_service_change_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_SEND_SERVICE_CHANGE_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

struct **gatts_set_attr_val_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_SET_ATTR_VAL_EVT.

Public Members

uint16_t **srvc_handle**

The service handle

uint16_t **attr_handle**

The attribute handle

esp_gatt_status_t **status**

Operation status

struct **gatts_start_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_START_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

struct **gatts_stop_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_STOP_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

struct **gatts_write_evt_param**

#include <esp_gatts_api.h> ESP_GATTS_WRITE_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint32_t **trans_id**

Transfer id

esp_bd_addr_t **bda**

The bluetooth device address which been written

uint16_t **handle**

The attribute handle

uint16_t **offset**

Offset of the value, if the value is too long

bool **need_rsp**

The write operation need to do response

bool **is_prep**

This write operation is prepare write

uint16_t **len**

The write attribute value length

uint8_t ***value**

The write attribute value

Macros

ESP_GATT_PREP_WRITE_CANCEL

Prepare write flag to indicate cancel prepare write

ESP_GATT_PREP_WRITE_EXEC

Prepare write flag to indicate execute prepare write

Type Definitions

typedef void (***esp_gatts_cb_t**)(*esp_gatts_cb_event_t* event, *esp_gatt_if_t* gatts_if, *esp_ble_gatts_cb_param_t* *param)

GATT Server callback function type.

Param event : Event type
Param gatts_if : GATT server access interface, normally different gatts_if correspond to different profile
Param param : Point to callback parameter, currently is union type

Enumerations

enum **esp_gatts_cb_event_t**

GATT Server callback function events.

Values:

enumerator **ESP_GATTS_REG_EVT**

When register application id, the event comes

enumerator **ESP_GATTS_READ_EVT**

When gatt client request read operation, the event comes

enumerator **ESP_GATTS_WRITE_EVT**

When gatt client request write operation, the event comes

enumerator **ESP_GATTS_EXEC_WRITE_EVT**

When gatt client request execute write, the event comes

enumerator **ESP_GATTS_MTU_EVT**

When set mtu complete, the event comes

enumerator **ESP_GATTS_CONF_EVT**

When receive confirm, the event comes

enumerator **ESP_GATTS_UNREG_EVT**

When unregister application id, the event comes

enumerator **ESP_GATTS_CREATE_EVT**

When create service complete, the event comes

enumerator **ESP_GATTS_ADD_INCL_SRVC_EVT**

When add included service complete, the event comes

enumerator **ESP_GATTS_ADD_CHAR_EVT**

When add characteristic complete, the event comes

enumerator **ESP_GATTS_ADD_CHAR_DESCR_EVT**

When add descriptor complete, the event comes

enumerator **ESP_GATTS_DELETE_EVT**

When delete service complete, the event comes

enumerator **ESP_GATTS_START_EVT**

When start service complete, the event comes

enumerator **ESP_GATTS_STOP_EVT**

When stop service complete, the event comes

enumerator **ESP_GATTS_CONNECT_EVT**

When gatt client connect, the event comes

enumerator **ESP_GATTS_DISCONNECT_EVT**

When gatt client disconnect, the event comes

enumerator **ESP_GATTS_OPEN_EVT**

When connect to peer, the event comes

enumerator **ESP_GATTS_CANCEL_OPEN_EVT**

When disconnect from peer, the event comes

enumerator **ESP_GATTS_CLOSE_EVT**

When gatt server close, the event comes

enumerator **ESP_GATTS_LISTEN_EVT**

When gatt listen to be connected the event comes

enumerator **ESP_GATTS_CONGEST_EVT**

When congest happen, the event comes

enumerator **ESP_GATTS_RESPONSE_EVT**

When gatt send response complete, the event comes

enumerator **ESP_GATTS_CREAT_ATTR_TAB_EVT**

When gatt create table complete, the event comes

enumerator **ESP_GATTS_SET_ATTR_VAL_EVT**

When gatt set attr value complete, the event comes

enumerator **ESP_GATTS_SEND_SERVICE_CHANGE_EVT**

When gatt send service change indication complete, the event comes

GATT CLIENT API

Application Example Check bluetooth/bluedroid/ble folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a GATT client demo and its tutorial. This demo can scan for devices, connect to the GATT server and discover its services.
  - bluetooth/bluedroid/ble/gatt_client
  - GATT Client Example Walkthrough
- This is a multiple connection demo and its tutorial. This demo can connect to multiple GATT server devices and discover their services.
  - bluetooth/bluedroid/ble/gattc_multi_connect
  - GATT Client Multi-connection Example Walkthrough
- This is a BLE SPP-Like demo. This demo, which acts as a GATT client, can receive data from UART and then send the data to the peer device automatically.
  - bluetooth/bluedroid/ble/ble_spp_client

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_gattc_api.h

Functions

esp_err_t **esp_ble_gattc_register_callback**(*esp_gattc_cb_t* callback)

This function is called to register application callbacks with GATTC module.

Parameters **callback** –**[in]** : pointer to the application callback function.

Returns

- ESP_OK: success
- other: failed

esp_gattc_cb_t **esp_ble_gattc_get_callback**(void)

This function is called to get the current application callbacks with BTA GATTC module.

Returns

- esp_gattC_cb_t : current callback

esp_err_t **esp_ble_gattc_app_register**(uint16_t app_id)

This function is called to register application callbacks with GATTC module.

Parameters **app_id** –**[in]** : Application Identify (UUID), for different application

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_app_unregister**(*esp_gatt_if_t* gattc_if)

This function is called to unregister an application from GATTC module.

Parameters **gattc_if** –**[in]** Gatt client access interface.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_open**(*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* remote_bda, *esp_ble_addr_type_t* remote_addr_type, bool is_direct)

Open a direct connection or add a background auto connection.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **remote_bda** –**[in]** remote device bluetooth device address.
- **remote_addr_type** –**[in]** remote device bluetooth device the address type.
- **is_direct** –**[in]** direct connection or background auto connection(by now, background auto connection is not supported).

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_aux_open**(*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* remote_bda, *esp_ble_addr_type_t* remote_addr_type, bool is_direct)

esp_err_t **esp_ble_gattc_close**(*esp_gatt_if_t* gattc_if, uint16_t conn_id)

Close the virtual connection to the GATT server. gattc may have multiple virtual GATT server connections when multiple app_id registered, this API only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. if you want to disconnect the physical connection directly, you can use esp_ble_gap_disconnect(esp_bd_addr_t remote_device).

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID to be closed.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_send_mtu_req**(*esp_gatt_if_t* gattc_if, uint16_t conn_id)

Configure the MTU size in the GATT channel. This can be done only once per connection. Before using, use esp_ble_gatt_set_local_mtu() to configure the local MTU size.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_search_service**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, *esp_bt_uuid_t* *filter_uuid)

This function is called to get service from local cache. This function report service search result by a callback event, and followed by a service search complete event. Note: 128-bit base UUID will automatically be converted to a 16-bit UUID in the search results. Other types of UUID remain unchanged.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID.
- **filter_uuid** –**[in]** a UUID of the service application is interested in. If Null, discover for all services.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_service**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, *esp_bt_uuid_t* *svc_uuid, *esp_gattc_service_elem_t* *result, uint16_t *count, uint16_t offset)

Find all the service with the given service uuid in the gattc cache, if the svc_uuid is NULL, find all the service. Note: It just get service from local cache, won't get from remote devices. If want to get it from remote device, need to used the esp_ble_gattc_cache_refresh, then call esp_ble_gattc_get_service again.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **svc_uuid** –**[in]** the pointer to the service uuid.
- **result** –**[out]** The pointer to the service which has been found in the gattc cache.
- **count** –**[inout]** input the number of service want to find, it will output the number of service has been found in the gattc cache with the given service uuid.
- **offset** –**[in]** Offset of the service position to get.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_all_char**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_gattc_char_elem_t* *result, uint16_t *count, uint16_t offset)

Find all the characteristic with the given service in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **start_handle** –**[in]** the attribute start handle.
- **end_handle** –**[in]** the attribute end handle
- **result** –**[out]** The pointer to the characteristic in the service.
- **count** –**[inout]** input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.
- **offset** –**[in]** Offset of the characteristic position to get.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_all_descr**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t char_handle, *esp_gattc_descr_elem_t* *result, uint16_t *count, uint16_t offset)

Find all the descriptor with the given characteristic in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **char_handle** –**[in]** the given characteristic handle
- **result** –**[out]** The pointer to the descriptor in the characteristic.
- **count** –**[inout]** input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.
- **offset** –**[in]** Offset of the descriptor position to get.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_char_by_uuid**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_bt_uuid_t* char_uuid, *esp_gattc_char_elem_t* *result, uint16_t *count)

Find the characteristic with the given characteristic uuid in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **start_handle** –**[in]** the attribute start handle
- **end_handle** –**[in]** the attribute end handle
- **char_uuid** –**[in]** the characteristic uuid
- **result** –**[out]** The pointer to the characteristic in the service.
- **count** –**[inout]** input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_descr_by_uuid**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_bt_uuid_t* char_uuid, *esp_bt_uuid_t* descr_uuid, *esp_gattc_descr_elem_t* *result, uint16_t *count)

Find the descriptor with the given characteristic uuid in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **start_handle** –**[in]** the attribute start handle
- **end_handle** –**[in]** the attribute end handle
- **char_uuid** –**[in]** the characteristic uuid.
- **descr_uuid** –**[in]** the descriptor uuid.
- **result** –**[out]** The pointer to the descriptor in the given characteristic.
- **count** –**[inout]** input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_descr_by_char_handle**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t char_handle, *esp_bt_uuid_t* descr_uuid, *esp_gattc_descr_elem_t* *result, uint16_t *count)

Find the descriptor with the given characteristic handle in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **char_handle** –**[in]** the characteristic handle.
- **descr_uuid** –**[in]** the descriptor uuid.
- **result** –**[out]** The pointer to the descriptor in the given characteristic.
- **count** –**[inout]** input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_include_service**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_bt_uuid_t* *incl_uuid, *esp_gattc_incl_svc_elem_t* *result, uint16_t *count)

Find the include service with the given service handle in the gattc cache Note: It just get include service from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **start_handle** –**[in]** the attribute start handle
- **end_handle** –**[in]** the attribute end handle
- **incl_uuid** –**[in]** the include service uuid
- **result** –**[out]** The pointer to the include service in the given service.
- **count** –**[inout]** input the number of include service want to find, it will output the number of include service has been found in the gattc cache with the given service.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_attr_count**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, *esp_gatt_db_attr_type_t* type, uint16_t start_handle, uint16_t end_handle, uint16_t char_handle, uint16_t *count)

Find the attribute count with the given service or characteristic in the gattc cache.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** connection ID which identify the server.
- **type** –**[in]** the attribute type.
- **start_handle** –**[in]** the attribute start handle, if the type is ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore
- **end_handle** –**[in]** the attribute end handle, if the type is ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore
- **char_handle** –**[in]** the characteristic handle, this parameter valid when the type is ESP_GATT_DB_DESCRIPTOR. If the type isn't ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore.
- **count** –**[out]** output the number of attribute has been found in the gattc cache with the given attribute type.

Returns

- ESP_OK: success
- other: failed

esp_gatt_status_t **esp_ble_gattc_get_db**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_gattc_db_elem_t* *db, uint16_t *count)

This function is called to get the GATT database. Note: It just get attribute data base from local cache, won't get from remote devices.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **start_handle** –**[in]** the attribute start handle
- **end_handle** –**[in]** the attribute end handle
- **conn_id** –**[in]** connection ID which identify the server.
- **db** –**[in]** output parameter which will contain the GATT database copy. Caller is responsible for freeing it.
- **count** –**[in]** number of elements in database.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_read_char**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, *esp_gatt_auth_req_t* auth_req)

This function is called to read a service' s characteristics of the given characteristic handle.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **handle** –**[in]** : characteritic handle to read.
- **auth_req** –**[in]** : authenticate request type

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_read_by_type**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, *esp_bt_uuid_t* *uuid, *esp_gatt_auth_req_t* auth_req)

This function is called to read a service' s characteristics of the given characteristic UUID.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **start_handle** –**[in]** : the attribute start handle.
- **end_handle** –**[in]** : the attribute end handle
- **uuid** –**[in]** : The UUID of attribute which will be read.
- **auth_req** –**[in]** : authenticate request type

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_read_multiple**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, *esp_gattc_multi_t* *read_multi, *esp_gatt_auth_req_t* auth_req)

This function is called to read multiple characteristic or characteristic descriptors.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **read_multi** –**[in]** : pointer to the read multiple parameter.
- **auth_req** –**[in]** : authenticate request type

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_read_multiple_variable**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, *esp_gattc_multi_t* *read_multi, *esp_gatt_auth_req_t* auth_req)

This function is called to read multiple variable length characteristic or characteristic descriptors.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **read_multi** –**[in]** : pointer to the read multiple parameter.
- **auth_req** –**[in]** : authenticate request type

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_read_char_descr**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, *esp_gatt_auth_req_t* auth_req)

This function is called to read a characteristics descriptor.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **handle** –**[in]** : descriptor handle to read.
- **auth_req** –**[in]** : authenticate request type

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_write_char**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, uint16_t value_len, uint8_t *value, *esp_gatt_write_type_t* write_type, *esp_gatt_auth_req_t* auth_req)

This function is called to write characteristic value.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **handle** –**[in]** : characteristic handle to write.
- **value_len** –**[in]** length of the value to be written.
- **value** –**[in]** : the value to be written.
- **write_type** –**[in]** : the type of attribute write operation.
- **auth_req** –**[in]** : authentication request.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_write_char_descr**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, uint16_t value_len, uint8_t *value, *esp_gatt_write_type_t* write_type, *esp_gatt_auth_req_t* auth_req)

This function is called to write characteristic descriptor value.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID
- **handle** –**[in]** : descriptor handle to write.
- **value_len** –**[in]** length of the value to be written.
- **value** –**[in]** : the value to be written.
- **write_type** –**[in]** : the type of attribute write operation.
- **auth_req** –**[in]** : authentication request.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_prepare_write**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, uint16_t offset, uint16_t value_len, uint8_t *value, *esp_gatt_auth_req_t* auth_req)

This function is called to prepare write a characteristic value.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **handle** –**[in]** : characteristic handle to prepare write.
- **offset** –**[in]** : offset of the write value.
- **value_len** –**[in]** length of the value to be written.
- **value** –**[in]** : the value to be written.
- **auth_req** –**[in]** : authentication request.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_prepare_write_char_descr**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, uint16_t handle, uint16_t offset, uint16_t value_len, uint8_t *value, *esp_gatt_auth_req_t* auth_req)

This function is called to prepare write a characteristic descriptor value.

Parameters
- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **handle** –**[in]** : characteristic descriptor handle to prepare write.
- **offset** –**[in]** : offset of the write value.
- **value_len** –**[in]** length of the value to be written.
- **value** –**[in]** : the value to be written.
- **auth_req** –**[in]** : authentication request.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_execute_write**(*esp_gatt_if_t* gattc_if, uint16_t conn_id, bool is_execute)

This function is called to execute write a prepare write sequence.

Parameters
- **gattc_if** –**[in]** Gatt client access interface.
- **conn_id** –**[in]** : connection ID.
- **is_execute** –**[in]** : execute or cancel.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_register_for_notify**(*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* server_bda, uint16_t handle)

This function is called to register for notification of a service.

Parameters
- **gattc_if** –**[in]** Gatt client access interface.
- **server_bda** –**[in]** : target GATT server.
- **handle** –**[in]** : GATT characteristic handle.

Returns
- ESP_OK: registration succeeds
- other: failed

esp_err_t **esp_ble_gattc_unregister_for_notify**(*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* server_bda, uint16_t handle)

This function is called to de-register for notification of a service.

Parameters
- **gattc_if** –**[in]** Gatt client access interface.
- **server_bda** –**[in]** : target GATT server.
- **handle** –**[in]** : GATT characteristic handle.

Returns
- ESP_OK: unregister succeeds
- other: failed

esp_err_t **esp_ble_gattc_cache_refresh**(*esp_bd_addr_t* remote_bda)

Refresh the server cache store in the gattc stack of the remote device. If the device is connected, this API will restart the discovery of service information of the remote device.

Parameters **remote_bda** –**[in]** remote device BD address.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_cache_assoc** (*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* src_addr, *esp_bd_addr_t* assoc_addr, bool is_assoc)

Add or delete the associated address with the source address. Note: The role of this API is mainly when the client side has stored a server-side database, when it needs to connect another device, but the device ' s attribute database is the same as the server database stored on the client-side, calling this API can use the database that the device has stored used as the peer server database to reduce the attribute database search and discovery process and speed up the connection time. The associated address mains that device want to used the database has stored in the local cache. The source address mains that device want to share the database to the associated address device.

Parameters

- **gattc_if** –**[in]** Gatt client access interface.
- **src_addr** –**[in]** the source address which provide the attribute table.
- **assoc_addr** –**[in]** the associated device address which went to share the attribute table with the source address.
- **is_assoc** –**[in]** true add the associated device address, false remove the associated device address.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_cache_get_addr_list** (*esp_gatt_if_t* gattc_if)

Get the address list which has store the attribute table in the gattc cache. There will callback ESP_GATTC_GET_ADDR_LIST_EVT event when get address list complete.

Parameters **gattc_if** –**[in]** Gatt client access interface.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_ble_gattc_cache_clean** (*esp_bd_addr_t* remote_bda)

Clean the service cache of this device in the gattc stack,.

Parameters **remote_bda** –**[in]** remote device BD address.

Returns

- ESP_OK: success
- other: failed

Unions

union **esp_ble_gattc_cb_param_t**

#include <esp_gattc_api.h> Gatt client callback parameters union.

Public Members

struct *esp_ble_gattc_cb_param_t*::*gattc_reg_evt_param* **reg**

Gatt client callback param of ESP_GATTC_REG_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_open_evt_param* **open**

Gatt client callback param of ESP_GATTC_OPEN_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_close_evt_param* **close**

Gatt client callback param of ESP_GATTC_CLOSE_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_cfg_mtu_evt_param* **cfg_mtu**

Gatt client callback param of ESP_GATTC_CFG_MTU_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_search_cmpl_evt_param* **search_cmpl**
Gatt client callback param of ESP_GATTC_SEARCH_CMPL_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_search_res_evt_param* **search_res**
Gatt client callback param of ESP_GATTC_SEARCH_RES_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_read_char_evt_param* **read**
Gatt client callback param of ESP_GATTC_READ_CHAR_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_write_evt_param* **write**
Gatt client callback param of ESP_GATTC_WRITE_DESCR_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_exec_cmpl_evt_param* **exec_cmpl**
Gatt client callback param of ESP_GATTC_EXEC_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_notify_evt_param* **notify**
Gatt client callback param of ESP_GATTC_NOTIFY_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_srvc_chg_evt_param* **srvc_chg**
Gatt client callback param of ESP_GATTC_SRVC_CHG_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_congest_evt_param* **congest**
Gatt client callback param of ESP_GATTC_CONGEST_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_reg_for_notify_evt_param* **reg_for_notify**
Gatt client callback param of ESP_GATTC_REG_FOR_NOTIFY_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_unreg_for_notify_evt_param* **unreg_for_notify**
Gatt client callback param of ESP_GATTC_UNREG_FOR_NOTIFY_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_connect_evt_param* **connect**
Gatt client callback param of ESP_GATTC_CONNECT_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_disconnect_evt_param* **disconnect**
Gatt client callback param of ESP_GATTC_DISCONNECT_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_set_assoc_addr_cmp_evt_param* **set_assoc_cmp**
Gatt client callback param of ESP_GATTC_SET_ASSOC_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_get_addr_list_evt_param* **get_addr_list**
Gatt client callback param of ESP_GATTC_GET_ADDR_LIST_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_queue_full_evt_param* **queue_full**
Gatt client callback param of ESP_GATTC_QUEUE_FULL_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_dis_srvc_cmpl_evt_param* **dis_srvc_cmpl**
Gatt client callback param of ESP_GATTC_DIS_SRVC_CMPL_EVT

struct **gattc_cfg_mtu_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_CFG_MTU_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

uint16_t **mtu**
MTU size

struct **gattc_close_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_CLOSE_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

esp_gatt_conn_reason_t **reason**
The reason of gatt connection close

struct **gattc_congest_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_CONGEST_EVT.

Public Members

uint16_t **conn_id**
Connection id

bool **congested**
Congested or not

struct **gattc_connect_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_CONNECT_EVT.

Public Members

uint16_t **conn_id**
Connection id

uint8_t **link_role**

Link role : master role = 0 ; slave role = 1

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

esp_gatt_conn_params_t **conn_params**

current connection parameters

esp_ble_addr_type_t **ble_addr_type**

Remote BLE device address type

uint16_t **conn_handle**

HCI connection handle

struct **gattc_dis_srvc_cmpl_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_DIS_SRVC_CMPL_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

struct **gattc_disconnect_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_DISCONNECT_EVT.

Public Members

esp_gatt_conn_reason_t **reason**

disconnection reason

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

struct **gattc_exec_cmpl_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_EXEC_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

struct **gattc_get_addr_list_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_GET_ADDR_LIST_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint8_t **num_addr**

The number of address in the gattc cache address list

esp_bd_addr_t ***addr_list**

The pointer to the address list which has been get from the gattc cache

struct **gattc_notify_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_NOTIFY_EVT.

Public Members

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

uint16_t **handle**

The Characteristic or descriptor handle

uint16_t **value_len**

Notify attribute value

uint8_t ***value**

Notify attribute value

bool **is_notify**

True means notify, false means indicate

struct **gattc_open_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_OPEN_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

uint16_t **mtu**
MTU size

struct **gattc_queue_full_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_QUEUE_FULL_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

bool **is_full**
The gattc command queue is full or not

struct **gattc_read_char_evt_param**
#include <esp_gattc_api.h> ESP_GATTC_READ_CHAR_EVT, ESP_GATTC_READ_DESCR_EVT, ESP_GATTC_READ_MULTIPLE_EVT, ESP_GATTC_READ_MULTI_VAR_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

uint16_t **handle**
Characteristic handle

uint8_t ***value**
Characteristic value

uint16_t **value_len**
Characteristic value length

struct **gattc_reg_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_REG_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **app_id**

Application id which input in register API

struct **gattc_reg_for_notify_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_REG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **handle**

The characteristic or descriptor handle

struct **gattc_search_cmpl_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_SEARCH_CMPL_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

esp_service_source_t **searched_service_source**

The source of the service information

struct **gattc_search_res_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_SEARCH_RES_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint16_t **start_handle**

Service start handle

uint16_t **end_handle**

Service end handle

esp_gatt_id_t **srvc_id**

Service id, include service uuid and other information

bool **is_primary**

True if this is the primary service

struct **gattc_set_assoc_addr_cmp_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_SET_ASSOC_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

struct **gattc_srvc_chg_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_SRVC_CHG_EVT.

Public Members

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

struct **gattc_unreg_for_notify_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_UNREG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **handle**

The characteristic or descriptor handle

struct **gattc_write_evt_param**

#include <esp_gattc_api.h> ESP_GATTC_WRITE_CHAR_EVT, ESP_GATTC_PREP_WRITE_EVT, ESP_GATTC_WRITE_DESCR_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

uint16_t **handle**

The Characteristic or descriptor handle

uint16_t **offset**

The prepare write offset, this value is valid only when prepare write

Type Definitions

typedef void (***esp_gattc_cb_t**)(*esp_gattc_cb_event_t* event, *esp_gatt_if_t* gattc_if, *esp_ble_gattc_cb_param_t* *param)

GATT Client callback function type.

Param event : Event type
Param gattc_if : GATT client access interface, normally different gattc_if correspond to different profile
Param param : Point to callback parameter, currently is union type

Enumerations

enum **esp_gattc_cb_event_t**

GATT Client callback function events.

Values:

enumerator **ESP_GATTC_REG_EVT**

When GATT client is registered, the event comes

enumerator **ESP_GATTC_UNREG_EVT**

When GATT client is unregistered, the event comes

enumerator **ESP_GATTC_OPEN_EVT**

When GATT virtual connection is set up, the event comes

enumerator **ESP_GATTC_READ_CHAR_EVT**

When GATT characteristic is read, the event comes

enumerator **ESP_GATTC_WRITE_CHAR_EVT**

When GATT characteristic write operation completes, the event comes

enumerator **ESP_GATTC_CLOSE_EVT**

When GATT virtual connection is closed, the event comes

enumerator **ESP_GATTC_SEARCH_CMPL_EVT**

When GATT service discovery is completed, the event comes

enumerator **ESP_GATTC_SEARCH_RES_EVT**

When GATT service discovery result is got, the event comes

enumerator **ESP_GATTC_READ_DESCR_EVT**

When GATT characteristic descriptor read completes, the event comes

enumerator **ESP_GATTC_WRITE_DESCR_EVT**

When GATT characteristic descriptor write completes, the event comes

enumerator **ESP_GATTC_NOTIFY_EVT**

When GATT notification or indication arrives, the event comes

enumerator **ESP_GATTC_PREP_WRITE_EVT**

When GATT prepare-write operation completes, the event comes

enumerator **ESP_GATTC_EXEC_EVT**

When write execution completes, the event comes

enumerator **ESP_GATTC_ACL_EVT**

When ACL connection is up, the event comes

enumerator **ESP_GATTC_CANCEL_OPEN_EVT**

When GATT client ongoing connection is cancelled, the event comes

enumerator **ESP_GATTC_SRVC_CHG_EVT**

When "service changed" occurs, the event comes

enumerator **ESP_GATTC_ENC_CMPL_CB_EVT**

When encryption procedure completes, the event comes

enumerator **ESP_GATTC_CFG_MTU_EVT**

When configuration of MTU completes, the event comes

enumerator **ESP_GATTC_ADV_DATA_EVT**

When advertising of data, the event comes

enumerator **ESP_GATTC_MULT_ADV_ENB_EVT**

When multi-advertising is enabled, the event comes

enumerator **ESP_GATTC_MULT_ADV_UPD_EVT**

When multi-advertising parameters are updated, the event comes

enumerator **ESP_GATTC_MULT_ADV_DATA_EVT**

When multi-advertising data arrives, the event comes

enumerator **ESP_GATTC_MULT_ADV_DIS_EVT**

When multi-advertising is disabled, the event comes

enumerator **ESP_GATTC_CONGEST_EVT**

When GATT connection congestion comes, the event comes

enumerator **ESP_GATTC_BTH_SCAN_ENB_EVT**

When batch scan is enabled, the event comes

enumerator **ESP_GATTC_BTH_SCAN_CFG_EVT**

When batch scan storage is configured, the event comes

enumerator **ESP_GATTC_BTH_SCAN_RD_EVT**

When Batch scan read event is reported, the event comes

enumerator **ESP_GATTC_BTH_SCAN_THR_EVT**

When Batch scan threshold is set, the event comes

enumerator **ESP_GATTC_BTH_SCAN_PARAM_EVT**

When Batch scan parameters are set, the event comes

enumerator **ESP_GATTC_BTH_SCAN_DIS_EVT**

When Batch scan is disabled, the event comes

enumerator **ESP_GATTC_SCAN_FLT_CFG_EVT**

When Scan filter configuration completes, the event comes

enumerator **ESP_GATTC_SCAN_FLT_PARAM_EVT**

When Scan filter parameters are set, the event comes

enumerator **ESP_GATTC_SCAN_FLT_STATUS_EVT**

When Scan filter status is reported, the event comes

enumerator **ESP_GATTC_ADV_VSC_EVT**

When advertising vendor spec content event is reported, the event comes

enumerator **ESP_GATTC_REG_FOR_NOTIFY_EVT**

When register for notification of a service completes, the event comes

enumerator **ESP_GATTC_UNREG_FOR_NOTIFY_EVT**

When unregister for notification of a service completes, the event comes

enumerator **ESP_GATTC_CONNECT_EVT**

When the ble physical connection is set up, the event comes

enumerator **ESP_GATTC_DISCONNECT_EVT**

When the ble physical connection disconnected, the event comes

enumerator **ESP_GATTC_READ_MULTIPLE_EVT**

When the ble characteristic or descriptor multiple complete, the event comes

enumerator **ESP_GATTC_QUEUE_FULL_EVT**

When the gattc command queue full, the event comes

enumerator **ESP_GATTC_SET_ASSOC_EVT**

When the ble gattc set the associated address complete, the event comes

enumerator **ESP_GATTC_GET_ADDR_LIST_EVT**

When the ble get gattc address list in cache finish, the event comes

enumerator **ESP_GATTC_DIS_SRVC_CMPL_EVT**

When the ble discover service complete, the event comes

enumerator **ESP_GATTC_READ_MULTI_VAR_EVT**

When read multiple variable characteristic complete, the event comes

BLUFI API

Overview BLUFI is a profile based GATT to config ESP32 WIFI to connect/disconnect AP or setup a softap and etc. Use should concern these things:

1. The event sent from profile. Then you need to do something as the event indicate.
2. Security reference. You can write your own Security functions such as symmetrical encryption/decryption and checksum functions. Even you can define the "Key Exchange/Negotiation" procedure.

Application Example Check bluetooth folder in ESP-IDF examples, which contains the following application:

- This is the BLUFI demo. This demo can set ESP32's wifi to softap/station/softap&station mode and config wifi connections - bluetooth/blufi

API Reference

Header File

- components/bt/common/api/include/api/esp_blufi_api.h

Functions

esp_err_t **esp_blufi_register_callbacks**(*esp_blufi_callbacks_t* *callbacks)

This function is called to receive blufi callback event.

Parameters **callbacks** –**[in]** callback functions
Returns ESP_OK - success, other - failed

esp_err_t **esp_blufi_profile_init**(void)

This function is called to initialize blufi_profile.

Returns ESP_OK - success, other - failed

esp_err_t **esp_blufi_profile_deinit**(void)

This function is called to de-initialize blufi_profile.

Returns ESP_OK - success, other - failed

esp_err_t **esp_blufi_send_wifi_conn_report**(*wifi_mode_t* opmode, *esp_blufi_sta_conn_state_t* sta_conn_state, uint8_t softap_conn_num, *esp_blufi_extra_info_t* *extra_info)

This function is called to send wifi connection report.

Parameters

- **opmode** –: wifi opmode

- **sta_conn_state** –: station is already in connection or not
- **softap_conn_num** –: softap connection number
- **extra_info** –: extra information, such as sta_ssid, softap_ssid and etc.

Returns ESP_OK - success, other - failed

esp_err_t **esp_blufi_send_wifi_list**(uint16_t apCount, *esp_blufi_ap_record_t* *list)

This function is called to send wifi list.

Parameters

- **apCount** –: wifi list count
- **list** –: wifi list

Returns ESP_OK - success, other - failed

uint16_t **esp_blufi_get_version**(void)

Get BLUFI profile version.

Returns Most 8bit significant is Great version, Least 8bit is Sub version

esp_err_t **esp_blufi_send_error_info**(*esp_blufi_error_state_t* state)

This function is called to send blufi error information.

Parameters **state** –: error state

Returns ESP_OK - success, other - failed

esp_err_t **esp_blufi_send_custom_data**(uint8_t *data, uint32_t data_len)

This function is called to custom data.

Parameters

- **data** –: custom data value
- **data_len** –: the length of custom data

Returns ESP_OK - success, other - failed

Unions

union **esp_blufi_cb_param_t**

#include <esp_blufi_api.h> BLUFI callback parameters union.

Public Members

struct *esp_blufi_cb_param_t*::*blufi_init_finish_evt_param* **init_finish**

Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_deinit_finish_evt_param* **deinit_finish**

Blufi callback param of ESP_BLUFI_EVENT_DEINIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_set_wifi_mode_evt_param* **wifi_mode**

Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_connect_evt_param* **connect**

Blufi callback param of ESP_BLUFI_EVENT_CONNECT

struct *esp_blufi_cb_param_t*::*blufi_disconnect_evt_param* **disconnect**

Blufi callback param of ESP_BLUFI_EVENT_DISCONNECT

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_bssid_evt_param* **sta_bssid**

Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_BSSID

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_ssid_evt_param* **sta_ssid**

Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_SSID

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_passwd_evt_param* **sta_passwd**

Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_PASSWD

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_ssid_evt_param* **softap_ssid**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_SSID

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_passwd_evt_param* **softap_passwd**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_max_conn_num_evt_param* **softap_max_conn_num**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_auth_mode_evt_param* **softap_auth_mode**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_channel_evt_param* **softap_channel**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL

struct *esp_blufi_cb_param_t*::*blufi_recv_username_evt_param* **username**

Blufi callback param of ESP_BLUFI_EVENT_RECV_USERNAME

struct *esp_blufi_cb_param_t*::*blufi_recv_ca_evt_param* **ca**

Blufi callback param of ESP_BLUFI_EVENT_RECV_CA_CERT

struct *esp_blufi_cb_param_t*::*blufi_recv_client_cert_evt_param* **client_cert**

Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_CERT

struct *esp_blufi_cb_param_t*::*blufi_recv_server_cert_evt_param* **server_cert**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_CERT

struct *esp_blufi_cb_param_t*::*blufi_recv_client_pkey_evt_param* **client_pkey**

Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY

struct *esp_blufi_cb_param_t*::*blufi_recv_server_pkey_evt_param* **server_pkey**

Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

struct *esp_blufi_cb_param_t*::*blufi_get_error_evt_param* **report_error**

Blufi callback param of ESP_BLUFI_EVENT_REPORT_ERROR

struct *esp_blufi_cb_param_t*::*blufi_recv_custom_data_evt_param* **custom_data**

Blufi callback param of ESP_BLUFI_EVENT_RECV_CUSTOM_DATA

struct **blufi_connect_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_CONNECT.

Public Members

esp_blufi_bd_addr_t **remote_bda**

Blufi Remote bluetooth device address

uint8_t **server_if**

server interface

uint16_t **conn_id**

Connection id

struct **blufi_deinit_finish_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_DEINIT_FINISH.

Public Members

esp_blufi_deinit_state_t **state**

De-initial status

struct **blufi_disconnect_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_DISCONNECT.

Public Members

esp_blufi_bd_addr_t **remote_bda**

Blufi Remote bluetooth device address

struct **blufi_get_error_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_REPORT_ERROR.

Public Members

esp_blufi_error_state_t **state**

Blufi error state

struct **blufi_init_finish_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_INIT_FINISH.

Public Members

esp_blufi_init_state_t **state**

Initial status

struct **blufi_recv_ca_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CA_CERT.

Public Members

uint8_t *__cert__

CA certificate point

int __cert_len__

CA certificate length

struct __blufi_recv_client_cert_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_CERT

Public Members

uint8_t *__cert__

Client certificate point

int __cert_len__

Client certificate length

struct __blufi_recv_client_pkey_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY

Public Members

uint8_t *__pkey__

Client Private Key point, if Client certificate not contain Key

int __pkey_len__

Client Private key length

struct __blufi_recv_custom_data_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CUSTOM_DATA.

Public Members

uint8_t *__data__

Custom data

uint32_t __data_len__

Custom data Length

struct __blufi_recv_server_cert_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_CERT

Public Members

uint8_t *__cert__

Client certificate point

int __cert_len__

Client certificate length

struct __blufi_recv_server_pkey_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

Public Members

uint8_t *__pkey__

Client Private Key point, if Client certificate not contain Key

int __pkey_len__

Client Private key length

struct __blufi_recv_softap_auth_mode_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE.

Public Members

wifi_auth_mode_t __auth_mode__

Authentication mode

struct __blufi_recv_softap_channel_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL.

Public Members

uint8_t __channel__

Authentication mode

struct __blufi_recv_softap_max_conn_num_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM.

Public Members

int __max_conn_num__

SSID

struct __blufi_recv_softap_passwd_evt_param__

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD.

Public Members

uint8_t ***passwd**

Password

int **passwd_len**

Password Length

struct **blufi_recv_softap_ssid_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_SSID.

Public Members

uint8_t ***ssid**

SSID

int **ssid_len**

SSID length

struct **blufi_recv_sta_bssid_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_BSSID.

Public Members

uint8_t **bssid**[6]

BSSID

struct **blufi_recv_sta_passwd_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_PASSWD.

Public Members

uint8_t ***passwd**

Password

int **passwd_len**

Password Length

struct **blufi_recv_sta_ssid_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_SSID.

Public Members

uint8_t ***ssid**

SSID

int **ssid_len**

SSID length

struct **blufi_recv_username_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_USERNAME.

Public Members

uint8_t ***name**

Username point

int **name_len**

Username length

struct **blufi_set_wifi_mode_evt_param**

#include <esp_blufi_api.h> ESP_BLUFI_EVENT_SET_WIFI_MODE.

Public Members

wifi_mode_t **op_mode**

Wifi operation mode

Structures

struct **esp_blufi_extra_info_t**

BLUFI extra information structure.

Public Members

uint8_t **sta_bssid**[6]

BSSID of station interface

bool **sta_bssid_set**

is BSSID of station interface set

uint8_t ***sta_ssid**

SSID of station interface

int **sta_ssid_len**

length of SSID of station interface

uint8_t ***sta_passwd**

password of station interface

int **sta_passwd_len**

length of password of station interface

uint8_t ***softap_ssid**

SSID of softap interface

int **softap_ssid_len**

length of SSID of softap interface

uint8_t ***softap_passwd**

password of station interface

int **softap_passwd_len**

length of password of station interface

uint8_t **softap_authmode**

authentication mode of softap interface

bool **softap_authmode_set**

is authentication mode of softap interface set

uint8_t **softap_max_conn_num**

max connection number of softap interface

bool **softap_max_conn_num_set**

is max connection number of softap interface set

uint8_t **softap_channel**

channel of softap interface

bool **softap_channel_set**

is channel of softap interface set

uint8_t **sta_max_conn_retry**

max retry of sta establish connection

bool **sta_max_conn_retry_set**

is max retry of sta establish connection set

uint8_t **sta_conn_end_reason**

reason of sta connection end

bool **sta_conn_end_reason_set**

is reason of sta connection end set

int8_t **sta_conn_rssi**

rssi of sta connection

bool **sta_conn_rssi_set**

is rssi of sta connection set

struct **esp_blufi_ap_record_t**

Description of an WiFi AP.

Public Members

uint8_t **ssid**[33]

SSID of AP

int8_t **rssi**

signal strength of AP

struct **esp_blufi_callbacks_t**

BLUFI callback functions type.

Public Members

esp_blufi_event_cb_t **event_cb**

BLUFI event callback

esp_blufi_negotiate_data_handler_t **negotiate_data_handler**

BLUFI negotiate data function for negotiate share key

esp_blufi_encrypt_func_t **encrypt_func**

BLUFI encrypt data function with share key generated by negotiate_data_handler

esp_blufi_decrypt_func_t **decrypt_func**

BLUFI decrypt data function with share key generated by negotiate_data_handler

esp_blufi_checksum_func_t **checksum_func**

BLUFI check sum function (FCS)

Macros

ESP_BLUFI_BD_ADDR_LEN

Bluetooth address length.

Type Definitions

typedef uint8_t **esp_blufi_bd_addr_t**[ESP_BLUFI_BD_ADDR_LEN]

Bluetooth device address.

typedef void (***esp_blufi_event_cb_t**)(*esp_blufi_cb_event_t* event, *esp_blufi_cb_param_t* *param)

BLUFI event callback function type.

Param event : Event type
Param param : Point to callback parameter, currently is union type

typedef void (***esp_blufi_negotiate_data_handler_t**)(uint8_t *data, int len, uint8_t **output_data, int *output_len, bool *need_free)

BLUFI negotiate data handler.

Param data : data from phone
Param len : length of data from phone
Param output_data : data want to send to phone

Param output_len : length of data want to send to phone
Param need_free : output reporting if memory needs to be freed or not *

typedef int (***esp_blufi_encrypt_func_t**)(uint8_t iv8, uint8_t *crypt_data, int crypt_len)

BLUFI encrypt the data after negotiate a share key.

Param iv8 : initial vector(8bit), normally, blufi core will input packet sequence number
Param crypt_data : plain text and encrypted data, the encrypt function must support autochthonous encrypt
Param crypt_len : length of plain text
Return Nonnegative number is encrypted length, if error, return negative number;

typedef int (***esp_blufi_decrypt_func_t**)(uint8_t iv8, uint8_t *crypt_data, int crypt_len)

BLUFI decrypt the data after negotiate a share key.

Param iv8 : initial vector(8bit), normally, blufi core will input packet sequence number
Param crypt_data : encrypted data and plain text, the encrypt function must support autochthonous decrypt
Param crypt_len : length of encrypted text
Return Nonnegative number is decrypted length, if error, return negative number;

typedef uint16_t (***esp_blufi_checksum_func_t**)(uint8_t iv8, uint8_t *data, int len)

BLUFI checksum.

Param iv8 : initial vector(8bit), normally, blufi core will input packet sequence number
Param data : data need to checksum
Param len : length of data

Enumerations

enum **esp_blufi_cb_event_t**

Values:

enumerator **ESP_BLUFI_EVENT_INIT_FINISH**

enumerator **ESP_BLUFI_EVENT_DEINIT_FINISH**

enumerator **ESP_BLUFI_EVENT_SET_WIFI_OPMODE**

enumerator **ESP_BLUFI_EVENT_BLE_CONNECT**

enumerator **ESP_BLUFI_EVENT_BLE_DISCONNECT**

enumerator **ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP**

enumerator **ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP**

enumerator **ESP_BLUFI_EVENT_GET_WIFI_STATUS**

enumerator **ESP_BLUFI_EVENT_DEAUTHENTICATE_STA**

enumerator **ESP_BLUFI_EVENT_RECV_STA_BSSID**

enumerator **ESP_BLUFI_EVENT_RECV_STA_SSID**

enumerator **ESP_BLUFI_EVENT_RECV_STA_PASSWD**

enumerator **ESP_BLUFI_EVENT_RECV_SOFTAP_SSID**

enumerator **ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD**

enumerator **ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM**

enumerator **ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE**

enumerator **ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL**

enumerator **ESP_BLUFI_EVENT_RECV_USERNAME**

enumerator **ESP_BLUFI_EVENT_RECV_CA_CERT**

enumerator **ESP_BLUFI_EVENT_RECV_CLIENT_CERT**

enumerator **ESP_BLUFI_EVENT_RECV_SERVER_CERT**

enumerator **ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY**

enumerator **ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY**

enumerator **ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE**

enumerator **ESP_BLUFI_EVENT_GET_WIFI_LIST**

enumerator **ESP_BLUFI_EVENT_REPORT_ERROR**

enumerator **ESP_BLUFI_EVENT_RECV_CUSTOM_DATA**

enum **esp_blufi_sta_conn_state_t**

BLUFI config status.

Values:

enumerator **ESP_BLUFI_STA_CONN_SUCCESS**

enumerator **ESP_BLUFI_STA_CONN_FAIL**

enumerator **ESP_BLUFI_STA_CONNECTING**

enumerator **ESP_BLUFI_STA_NO_IP**

enum **esp_blufi_init_state_t**

BLUFI init status.

Values:

enumerator **ESP_BLUFI_INIT_OK**

enumerator **ESP_BLUFI_INIT_FAILED**

enum **esp_blufi_deinit_state_t**

BLUFI deinit status.

Values:

enumerator **ESP_BLUFI_DEINIT_OK**

enumerator **ESP_BLUFI_DEINIT_FAILED**

enum **esp_blufi_error_state_t**

Values:

enumerator **ESP_BLUFI_SEQUENCE_ERROR**

enumerator **ESP_BLUFI_CHECKSUM_ERROR**

enumerator **ESP_BLUFI_DECRYPT_ERROR**

enumerator **ESP_BLUFI_ENCRYPT_ERROR**

enumerator **ESP_BLUFI_INIT_SECURITY_ERROR**

enumerator **ESP_BLUFI_DH_MALLOC_ERROR**

enumerator **ESP_BLUFI_DH_PARAM_ERROR**

enumerator **ESP_BLUFI_READ_PARAM_ERROR**

enumerator **ESP_BLUFI_MAKE_PUBLIC_ERROR**

enumerator **ESP_BLUFI_DATA_FORMAT_ERROR**

enumerator **ESP_BLUFI_CALC_MD5_ERROR**

enumerator **ESP_BLUFI_WIFI_SCAN_FAIL**

enumerator **ESP_BLUFI_MSG_STATE_ERROR**

2.3.3 CLASSIC BT

CLASSIC BLUETOOTH GAP API

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_gap_bt_api.h

Functions

static inline uint32_t **esp_bt_gap_get_cod_srvc**(uint32_t cod)

get major service field of COD

Parameters **cod** –**[in]** Class of Device
Returns major service bits

static inline uint32_t **esp_bt_gap_get_cod_major_dev**(uint32_t cod)

get major device field of COD

Parameters **cod** –**[in]** Class of Device
Returns major device bits

static inline uint32_t **esp_bt_gap_get_cod_minor_dev**(uint32_t cod)

get minor service field of COD

Parameters **cod** –**[in]** Class of Device
Returns minor service bits

static inline uint32_t **esp_bt_gap_get_cod_format_type**(uint32_t cod)

get format type of COD

Parameters **cod** –**[in]** Class of Device
Returns format type

static inline bool **esp_bt_gap_is_valid_cod**(uint32_t cod)

decide the integrity of COD

Parameters **cod** –**[in]** Class of Device
Returns
- true if cod is valid
- false otherise

esp_err_t **esp_bt_gap_register_callback**(*esp_bt_gap_cb_t* callback)

register callback function. This function should be called after esp_bluedroid_enable() completes successfully

Returns
- ESP_OK : Succeed
- ESP_FAIL: others

esp_err_t **esp_bt_gap_set_scan_mode**(*esp_bt_connection_mode_t* c_mode, *esp_bt_discovery_mode_t* d_mode)

Set discoverability and connectability mode for legacy bluetooth. This function should be called after esp_bluedroid_enable() completes successfully.

Parameters
- **c_mode** –**[in]** : one of the enums of esp_bt_connection_mode_t
- **d_mode** –**[in]** : one of the enums of esp_bt_discovery_mode_t

Returns
- ESP_OK : Succeed
- ESP_ERR_INVALID_ARG: if argument invalid
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled

- ESP_FAIL: others

esp_err_t **esp_bt_gap_start_discovery**(*esp_bt_inq_mode_t* mode, uint8_t inq_len, uint8_t num_rsps)

This function starts Inquiry and Name Discovery. This function should be called after esp_bluedroid_enable() completes successfully. When Inquiry is halted and cached results do not contain device name, then Name Discovery will connect to the peer target to get the device name. esp_bt_gap_cb_t will be called with ESP_BT_GAP_DISC_STATE_CHANGED_EVT when Inquiry is started or Name Discovery is completed. esp_bt_gap_cb_t will be called with ESP_BT_GAP_DISC_RES_EVT each time the two types of discovery results are got.

Parameters

- **mode** –**[in]** - Inquiry mode
- **inq_len** –**[in]** - Inquiry duration in 1.28 sec units, ranging from 0x01 to 0x30. This parameter only specifies the total duration of the Inquiry process,
  - when this time expires, Inquiry will be halted.
- **num_rsps** –**[in]** - Number of responses that can be received before the Inquiry is halted, value 0 indicates an unlimited number of responses.

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if invalid parameters are provided
- ESP_FAIL: others

esp_err_t **esp_bt_gap_cancel_discovery**(void)

Cancel Inquiry and Name Discovery. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will be called with ESP_BT_GAP_DISC_STATE_CHANGED_EVT if Inquiry or Name Discovery is cancelled by calling this function.

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_bt_gap_get_remote_services**(*esp_bd_addr_t* remote_bda)

Start SDP to get remote services. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will be called with ESP_BT_GAP_RMT_SRVCS_EVT after service discovery ends.

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_bt_gap_get_remote_service_record**(*esp_bd_addr_t* remote_bda, *esp_bt_uuid_t* *uuid)

Start SDP to look up the service matching uuid on the remote device. This function should be called after esp_bluedroid_enable() completes successfully.

esp_bt_gap_cb_t will be called with ESP_BT_GAP_RMT_SRVC_REC_EVT after service discovery ends

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

uint8_t ***esp_bt_gap_resolve_eir_data**(uint8_t *eir, *esp_bt_eir_type_t* type, uint8_t *length)

This function is called to get EIR data for a specific type.

Parameters

- **eir** –**[in]** - pointer of raw eir data to be resolved
- **type** –**[in]** - specific EIR data type
- **length** –**[out]** - return the length of EIR data excluding fields of length and data type

Returns pointer of starting position of eir data excluding eir data type, NULL if not found

esp_err_t **esp_bt_gap_config_eir_data**(*esp_bt_eir_data_t* *eir_data)

This function is called to config EIR data.

```
            esp_bt_gap_cb_t will be called with ESP_BT_GAP_CONFIG_EIR_DATA_
↪EVT after config EIR ends.
```

Parameters **eir_data** –**[in]** - pointer of EIR data content

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if param is invalid
- ESP_FAIL: others

esp_err_t **esp_bt_gap_set_cod**(*esp_bt_cod_t* cod, *esp_bt_cod_mode_t* mode)

This function is called to set class of device. The structure esp_bt_gap_cb_t will be called with ESP_BT_GAP_SET_COD_EVT after set COD ends. This function should be called after Bluetooth profiles are initialized, otherwise the user configured class of device can be overwritten. Some profiles have special restrictions on class of device, and changes may make these profiles unable to work.

Parameters

- **cod** –**[in]** - class of device
- **mode** –**[in]** - setting mode

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if param is invalid
- ESP_FAIL: others

esp_err_t **esp_bt_gap_get_cod**(*esp_bt_cod_t* *cod)

This function is called to get class of device.

Parameters **cod** –**[out]** - class of device

Returns

- ESP_OK : Succeed
- ESP_FAIL: others

esp_err_t **esp_bt_gap_read_rssi_delta**(*esp_bd_addr_t* remote_addr)

This function is called to read RSSI delta by address after connected. The RSSI value returned by ESP_BT_GAP_READ_RSSI_DELTA_EVT.

Parameters **remote_addr** –**[in]** - remote device address, corresponding to a certain connection handle

Returns

- ESP_OK : Succeed
- ESP_FAIL: others

esp_err_t **esp_bt_gap_remove_bond_device**(*esp_bd_addr_t* bd_addr)

Removes a device from the security database list of peer device.

Parameters **bd_addr** –**[in]** : BD address of the peer device

Returns - ESP_OK : success

- ESP_FAIL : failed

int **esp_bt_gap_get_bond_device_num**(void)

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

Returns - >= 0 : bonded devices number

- ESP_FAIL : failed

esp_err_t **esp_bt_gap_get_bond_device_list**(int *dev_num, *esp_bd_addr_t* *dev_list)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Parameters

- **dev_num** –**[inout]** Indicate the dev_list array(buffer) size as input. If dev_num is large enough, it means the actual number as output. Suggest that dev_num value equal to esp_ble_get_bond_device_num().
- **dev_list** –**[out]** an array(buffer) of `esp_bd_addr_t` type. Use for storing the bonded devices address. The dev_list should be allocated by who call this API.

Returns

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_bt_gap_set_pin**(*esp_bt_pin_type_t* pin_type, uint8_t pin_code_len, *esp_bt_pin_code_t* pin_code)

Set pin type and default pin code for legacy pairing.

Parameters

- **pin_type** –**[in]** Use variable or fixed pin. If pin_type is ESP_BT_PIN_TYPE_VARIABLE, pin_code and pin_code_len will be ignored, and ESP_BT_GAP_PIN_REQ_EVT will come when control requests for pin code. Else, will use fixed pin code and not callback to users.
- **pin_code_len** –**[in]** Length of pin_code
- **pin_code** –**[in]** Pin_code

Returns - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_pin_reply**(*esp_bd_addr_t* bd_addr, bool accept, uint8_t pin_code_len, *esp_bt_pin_code_t* pin_code)

Reply the pin_code to the peer device for legacy pairing when ESP_BT_GAP_PIN_REQ_EVT is coming.

Parameters

- **bd_addr** –**[in]** BD address of the peer
- **accept** –**[in]** Pin_code reply successful or declined.
- **pin_code_len** –**[in]** Length of pin_code
- **pin_code** –**[in]** Pin_code

Returns - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_set_security_param**(*esp_bt_sp_param_t* param_type, void *value, uint8_t len)

Set a GAP security parameter value. Overrides the default value.

Parameters

- **param_type** –**[in]** : the type of the param which is to be set
- **value** –**[in]** : the param value
- **len** –**[in]** : the length of the param value

Returns - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_ssp_passkey_reply**(*esp_bd_addr_t* bd_addr, bool accept, uint32_t passkey)

Reply the key value to the peer device in the legacy connection stage.

Parameters

- **bd_addr** –**[in]** : BD address of the peer
- **accept** –**[in]** : passkey entry successful or declined.

- **passkey** –**[in]** : passkey value, must be a 6 digit number, can be lead by 0.

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_ssp_confirm_reply**(*esp_bd_addr_t* bd_addr, bool accept)

Reply the confirm value to the peer device in the legacy connection stage.

Parameters
- **bd_addr** –**[in]** : BD address of the peer device
- **accept** –**[in]** : numbers to compare are the same or different

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_set_afh_channels**(*esp_bt_gap_afh_channels* channels)

Set the AFH channels.

Parameters **channels** –**[in]** : The n th such field (in the range 0 to 78) contains the value for channel n : 0 means channel n is bad. 1 means channel n is unknown. The most significant bit is reserved and shall be set to 0. At least 20 channels shall be marked as unknown.

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_read_remote_name**(*esp_bd_addr_t* remote_bda)

Read the remote device name.

Parameters **remote_bda** –**[in]** The remote device’ s address

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_set_qos**(*esp_bd_addr_t* remote_bda, uint32_t t_poll)

Config Quality of service.

Parameters
- **remote_bda** –**[in]** The remote device’ s address
- **t_poll** –**[in]** Poll interval, the maximum time between transmissions which from the master to a particular slave on the ACL logical transport. unit is 0.625ms

Returns - ESP_OK : success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

esp_err_t **esp_bt_gap_set_acl_pkt_types**(*esp_bd_addr_t* remote_bda, *esp_bt_acl_pkt_type_t* pkt_types)

Set ACL packet types. FOR INTERNAL TESTING ONLY. An ESP_BT_GAP_SET_ACL_PPKT_TYPES_EVT event will reported to the APP layer.

Returns - ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other: failed

Unions

union **esp_bt_gap_cb_param_t**

#include <esp_gap_bt_api.h> GAP state callback parameters.

Public Members

struct *esp_bt_gap_cb_param_t*::*disc_res_param* **disc_res**

discovery result parameter struct

struct *esp_bt_gap_cb_param_t*::*disc_state_changed_param* **disc_st_chg**

discovery state changed parameter struct

struct *esp_bt_gap_cb_param_t*::*rmt_srvcs_param* **rmt_srvcs**

services of remote device parameter struct

struct *esp_bt_gap_cb_param_t*::*rmt_srvc_rec_param* **rmt_srvc_rec**

specific service record from remote device parameter struct

struct *esp_bt_gap_cb_param_t*::*read_rssi_delta_param* **read_rssi_delta**

read rssi parameter struct

struct *esp_bt_gap_cb_param_t*::*config_eir_data_param* **config_eir_data**

config EIR data

struct *esp_bt_gap_cb_param_t*::*auth_cmpl_param* **auth_cmpl**

authentication complete parameter struct

struct *esp_bt_gap_cb_param_t*::*pin_req_param* **pin_req**

pin request parameter struct

struct *esp_bt_gap_cb_param_t*::*cfm_req_param* **cfm_req**

confirm request parameter struct

struct *esp_bt_gap_cb_param_t*::*key_notif_param* **key_notif**

passkey notif parameter struct

struct *esp_bt_gap_cb_param_t*::*key_req_param* **key_req**

passkey request parameter struct

struct *esp_bt_gap_cb_param_t*::*set_afh_channels_param* **set_afh_channels**

set AFH channel parameter struct

struct *esp_bt_gap_cb_param_t*::*read_rmt_name_param* **read_rmt_name**

read Remote Name parameter struct

struct *esp_bt_gap_cb_param_t*::*mode_chg_param* **mode_chg**

mode change event parameter struct

struct *esp_bt_gap_cb_param_t*::*bt_remove_bond_dev_cmpl_evt_param* **remove_bond_dev_cmpl**

Event parameter of ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT

struct *esp_bt_gap_cb_param_t*::*qos_cmpl_param* **qos_cmpl**

QoS complete parameter struct

struct *esp_bt_gap_cb_param_t*::*set_acl_pkt_types_param* **set_acl_pkt_types**

set ACL packet types parameter struct

struct *esp_bt_gap_cb_param_t*::*acl_conn_cmpl_stat_param* **acl_conn_cmpl_stat**

ACL connection complete status parameter struct

struct *esp_bt_gap_cb_param_t*::*acl_disconn_cmpl_stat_param* **acl_disconn_cmpl_stat**

ACL disconnection complete status parameter struct

struct **acl_conn_cmpl_stat_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_ACL_CONN_CMPL_STAT_EVT.

Public Members

esp_bt_status_t **stat**

ACL connection status

uint16_t **handle**

ACL connection handle

esp_bd_addr_t **bda**

remote bluetooth device address

struct **acl_disconn_cmpl_stat_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_ACL_DISCONN_CMPL_STAT_EVT.

Public Members

esp_bt_status_t **reason**

ACL disconnection reason

uint16_t **handle**

ACL connection handle

esp_bd_addr_t **bda**

remote bluetooth device address

struct **auth_cmpl_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_AUTH_CMPL_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **stat**

authentication complete status

uint8_t **device_name**[ESP_BT_GAP_MAX_BDNAME_LEN + 1]

device name

struct **bt_remove_bond_dev_cmpl_evt_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **status**

Indicate the remove bond device operation success status

struct **cfm_req_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_CFM_REQ_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

uint32_t **num_val**

the numeric value for comparison.

struct **config_eir_data_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_CONFIG_EIR_DATA_EVT *.

Public Members

esp_bt_status_t **stat**

config EIR status: ESP_BT_STATUS_SUCCESS: config success ESP_BT_STATUS_EIR_TOO_LARGE: the EIR data is more than 240B. The EIR may not contain the whole data. others: failed

uint8_t **eir_type_num**

the number of EIR types in EIR type

esp_bt_eir_type_t **eir_type**[ESP_BT_EIR_TYPE_MAX_NUM]

EIR types in EIR type

struct **disc_res_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_RES_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

int **num_prop**

number of properties got

esp_bt_gap_dev_prop_t ***prop**

properties discovered from the new device

struct **disc_state_changed_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_STATE_CHANGED_EVT.

Public Members

esp_bt_gap_discovery_state_t **state**

discovery state

struct **key_notif_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_KEY_NOTIF_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

uint32_t **passkey**

the numeric value for passkey entry.

struct **key_req_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_KEY_REQ_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

struct **mode_chg_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_MODE_CHG_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_pm_mode_t **mode**

PM mode

struct **pin_req_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_PIN_REQ_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

bool **min_16_digit**

TRUE if the pin returned must be at least 16 digits

struct **qos_cmpl_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_QOS_CMPL_EVT.

Public Members

esp_bt_status_t **stat**

QoS status

esp_bd_addr_t **bda**

remote bluetooth device address

uint32_t **t_poll**

poll interval, the maximum time between transmissions which from the master to a particular slave on the ACL logical transport. unit is 0.625ms.

struct **read_rmt_name_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_READ_REMOTE_NAME_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **stat**

read Remote Name status

uint8_t **rmt_name**[ESP_BT_GAP_MAX_BDNAME_LEN + 1]

Remote device name

struct **read_rssi_delta_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_READ_RSSI_DELTA_EVT *.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **stat**

read rssi status

int8_t **rssi_delta**

rssi delta value range -128 ~127, The value zero indicates that the RSSI is inside the Golden Receive Power Range, the Golden Receive Power Range is from ESP_BT_GAP_RSSI_LOW_THRLD to ESP_BT_GAP_RSSI_HIGH_THRLD

struct **rmt_srvc_rec_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVC_REC_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **stat**

service search status

struct **rmt_srvcs_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVCS_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

esp_bt_status_t **stat**

service search status

int **num_uuids**

number of UUID in uuid_list

esp_bt_uuid_t ***uuid_list**

list of service UUIDs of remote device

struct **set_acl_pkt_types_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_ACL_PKT_TYPE_CHANGED_EVT.

Public Members

esp_bt_status_t **status**

set ACL packet types status

esp_bd_addr_t **bda**

remote bluetooth device address

uint16_t **pkt_types**

packet types successfully set

struct **set_afh_channels_param**

#include <esp_gap_bt_api.h> ESP_BT_GAP_SET_AFH_CHANNELS_EVT.

Public Members

esp_bt_status_t **stat**

set AFH channel status

Structures

struct **esp_bt_cod_t**

Class of device.

Public Members

uint32_t **reserved_2**

undefined

uint32_t **minor**

minor class

uint32_t **major**

major class

uint32_t **service**

service class

uint32_t **reserved_8**

undefined

struct **esp_bt_gap_dev_prop_t**

Bluetooth Device Property Descriptor.

Public Members

esp_bt_gap_dev_prop_type_t **type**

Device property type

int **len**

Device property value length

void ***val**

Device property value

struct **esp_bt_eir_data_t**

EIR data content, according to "Supplement to the Bluetooth Core Specification".

Public Members

bool **fec_required**

FEC is required or not, true by default

bool **include_txpower**

EIR data include TX power, false by default

bool **include_uuid**

EIR data include UUID, false by default

bool **include_name**

EIR data include device name, true by default

uint8_t **flag**

EIR flags, see ESP_BT_EIR_FLAG for details, EIR will not include flag if it is 0, 0 by default

uint16_t **manufacturer_len**

Manufacturer data length, 0 by default

uint8_t ***p_manufacturer_data**

Manufacturer data point

uint16_t **url_len**

URL length, 0 by default

uint8_t ***p_url**

URL point

Macros

ESP_BT_GAP_RSSI_HIGH_THRLD

RSSI threshold.

High RSSI threshold

ESP_BT_GAP_RSSI_LOW_THRLD

Low RSSI threshold

ESP_BT_GAP_AFH_CHANNELS_LEN

ESP_BT_GAP_MAX_BDNAME_LEN

Maximum bytes of Bluetooth device name.

ESP_BT_GAP_EIR_DATA_LEN

Maximum size of EIR Significant part.

ESP_BT_EIR_TYPE_FLAGS

Extended Inquiry Response data type.

Flag with information such as BR/EDR and LE support

ESP_BT_EIR_TYPE_INCMPL_16BITS_UUID

Incomplete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_16BITS_UUID

Complete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_32BITS_UUID

Incomplete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_32BITS_UUID

Complete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_128BITS_UUID

Incomplete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_128BITS_UUID

Complete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_SHORT_LOCAL_NAME

Shortened Local Name

ESP_BT_EIR_TYPE_CMPL_LOCAL_NAME

Complete Local Name

ESP_BT_EIR_TYPE_TX_POWER_LEVEL

Tx power level, value is 1 octet ranging from -127 to 127, unit is dBm

ESP_BT_EIR_TYPE_URL

Uniform resource identifier

ESP_BT_EIR_TYPE_MANU_SPECIFIC

Manufacturer specific data

ESP_BT_EIR_TYPE_MAX_NUM

MAX number of EIR type

ESP_BT_ACL_PKT_TYPES_MASK_DM1

ESP_BT_ACL_PKT_TYPES_MASK_DH1

ESP_BT_ACL_PKT_TYPES_MASK_DM3

ESP_BT_ACL_PKT_TYPES_MASK_DH3

ESP_BT_ACL_PKT_TYPES_MASK_DM5

ESP_BT_ACL_PKT_TYPES_MASK_DH5

ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH1

ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH1

ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH3

ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH3

ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH5

ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH5

ESP_BT_ACL_DM1_ONLY

ESP_BT_ACL_DH1_ONLY

ESP_BT_ACL_DM3_ONLY

ESP_BT_ACL_DH3_ONLY

ESP_BT_ACL_DM5_ONLY

ESP_BT_ACL_DH5_ONLY

ESP_BT_ACL_2_DH1_ONLY

ESP_BT_ACL_3_DH1_ONLY

ESP_BT_ACL_2_DH3_ONLY

ESP_BT_ACL_3_DH3_ONLY

ESP_BT_ACL_2_DH5_ONLY

ESP_BT_ACL_3_DH5_ONLY

ESP_BT_EIR_FLAG_LIMIT_DISC

ESP_BT_EIR_FLAG_GEN_DISC

ESP_BT_EIR_FLAG_BREDR_NOT_SPT

ESP_BT_EIR_FLAG_DMT_CONTROLLER_SPT

ESP_BT_EIR_FLAG_DMT_HOST_SPT

ESP_BT_EIR_MAX_LEN

ESP_BT_PIN_CODE_LEN

Max pin code length

ESP_BT_IO_CAP_OUT

ESP_BT_IO_CAP_IO

ESP_BT_IO_CAP_IN

ESP_BT_IO_CAP_NONE

ESP_BT_PM_MD_ACTIVE

Active mode

ESP_BT_PM_MD_HOLD

Hold mode

ESP_BT_PM_MD_SNIFF

Sniff mode

ESP_BT_PM_MD_PARK

Park state

ESP_BT_COD_SRVC_BIT_MASK

Bits of major service class field.

Major service bit mask

ESP_BT_COD_SRVC_BIT_OFFSET

Major service bit offset

ESP_BT_COD_MAJOR_DEV_BIT_MASK

Bits of major device class field.

Major device bit mask

ESP_BT_COD_MAJOR_DEV_BIT_OFFSET

Major device bit offset

ESP_BT_COD_MINOR_DEV_BIT_MASK

Bits of minor device class field.

Minor device bit mask

ESP_BT_COD_MINOR_DEV_BIT_OFFSET

Minor device bit offset

ESP_BT_COD_FORMAT_TYPE_BIT_MASK

Bits of format type.

Format type bit mask

ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET

Format type bit offset

ESP_BT_COD_FORMAT_TYPE_1

Class of device format type 1.

ESP_BT_GAP_MIN_INQ_LEN

Minimum and Maximum inquiry length Minimum inquiry duration, unit is 1.28s

ESP_BT_GAP_MAX_INQ_LEN

Maximum inquiry duration, unit is 1.28s

ESP_BT_GAP_TPOLL_MIN

Minimum, Default and Maximum poll interval Minimum poll interval, unit is 625 microseconds

ESP_BT_GAP_TPOLL_DFT

Default poll interval, unit is 625 microseconds

ESP_BT_GAP_TPOLL_MAX

Maximum poll interval, unit is 625 microseconds

Type Definitions

typedef uint8_t **esp_bt_gap_afh_channels**[ESP_BT_GAP_AFH_CHANNELS_LEN]

typedef uint8_t **esp_bt_eir_type_t**

typedef uint16_t **esp_bt_acl_pkt_type_t**

typedef uint8_t **esp_bt_pin_code_t**[ESP_BT_PIN_CODE_LEN]

Pin Code (upto 128 bits) MSB is 0

typedef uint8_t **esp_bt_io_cap_t**

Combination of the IO Capability

typedef uint8_t **esp_bt_pm_mode_t**

typedef void (***esp_bt_gap_cb_t**)(*esp_bt_gap_cb_event_t* event, *esp_bt_gap_cb_param_t* *param)

bluetooth GAP callback function type

Param event : Event type
Param param : Pointer to callback parameter

Enumerations

enum **esp_bt_cod_mode_t**

class of device settings

Values:

enumerator **ESP_BT_SET_COD_MAJOR_MINOR**

overwrite major, minor class

enumerator **ESP_BT_SET_COD_SERVICE_CLASS**

set the bits in the input, the current bit will remain

enumerator **ESP_BT_CLR_COD_SERVICE_CLASS**

clear the bits in the input, others will remain

enumerator **ESP_BT_SET_COD_ALL**

overwrite major, minor, set the bits in service class

enumerator **ESP_BT_INIT_COD**

overwrite major, minor, and service class

enum **esp_bt_connection_mode_t**

Discoverability and Connectability mode.

Values:

enumerator **ESP_BT_NON_CONNECTABLE**

Non-connectable

enumerator **ESP_BT_CONNECTABLE**

Connectable

enum **esp_bt_discovery_mode_t**

Values:

enumerator **ESP_BT_NON_DISCOVERABLE**

Non-discoverable

enumerator **ESP_BT_LIMITED_DISCOVERABLE**

Limited Discoverable

enumerator **ESP_BT_GENERAL_DISCOVERABLE**

General Discoverable

enum **esp_bt_gap_dev_prop_type_t**

Bluetooth Device Property type.

Values:

enumerator **ESP_BT_GAP_DEV_PROP_BDNAME**

Bluetooth device name, value type is int8_t []

enumerator **ESP_BT_GAP_DEV_PROP_COD**

Class of Device, value type is uint32_t

enumerator **ESP_BT_GAP_DEV_PROP_RSSI**

Received Signal strength Indication, value type is int8_t, ranging from -128 to 127

enumerator **ESP_BT_GAP_DEV_PROP_EIR**

Extended Inquiry Response, value type is uint8_t []

enum **esp_bt_cod_srvc_t**

Major service class field of Class of Device, mutiple bits can be set.

Values:

enumerator **ESP_BT_COD_SRVC_NONE**

None indicates an invalid value

enumerator **ESP_BT_COD_SRVC_LMTD_DISCOVER**

Limited Discoverable Mode

enumerator **ESP_BT_COD_SRVC_POSITIONING**

Positioning (Location identification)

enumerator **ESP_BT_COD_SRVC_NETWORKING**

Networking, e.g. LAN, Ad hoc

enumerator **ESP_BT_COD_SRVC_RENDERING**

Rendering, e.g. Printing, Speakers

enumerator **ESP_BT_COD_SRVC_CAPTURING**

Capturing, e.g. Scanner, Microphone

enumerator **ESP_BT_COD_SRVC_OBJ_TRANSFER**

Object Transfer, e.g. v-Inbox, v-Folder

enumerator **ESP_BT_COD_SRVC_AUDIO**

Audio, e.g. Speaker, Microphone, Headset service

enumerator **ESP_BT_COD_SRVC_TELEPHONY**

Telephony, e.g. Cordless telephony, Modem, Headset service

enumerator **ESP_BT_COD_SRVC_INFORMATION**

Information, e.g., WEB-server, WAP-server

enum **esp_bt_pin_type_t**

Values:

enumerator **ESP_BT_PIN_TYPE_VARIABLE**

Refer to BTM_PIN_TYPE_VARIABLE

enumerator **ESP_BT_PIN_TYPE_FIXED**

Refer to BTM_PIN_TYPE_FIXED

enum **esp_bt_sp_param_t**

Values:

enumerator **ESP_BT_SP_IOCAP_MODE**

Set IO mode

enum **esp_bt_cod_major_dev_t**

Major device class field of Class of Device.

Values:

enumerator **ESP_BT_COD_MAJOR_DEV_MISC**

Miscellaneous

enumerator **ESP_BT_COD_MAJOR_DEV_COMPUTER**

Computer

enumerator **ESP_BT_COD_MAJOR_DEV_PHONE**

Phone(cellular, cordless, pay phone, modem

enumerator **ESP_BT_COD_MAJOR_DEV_LAN_NAP**

LAN, Network Access Point

enumerator **ESP_BT_COD_MAJOR_DEV_AV**

Audio/Video(headset, speaker, stereo, video display, VCR

enumerator **ESP_BT_COD_MAJOR_DEV_PERIPHERAL**

Peripheral(mouse, joystick, keyboard)

enumerator **ESP_BT_COD_MAJOR_DEV_IMAGING**

Imaging(printer, scanner, camera, display

enumerator **ESP_BT_COD_MAJOR_DEV_WEARABLE**

Wearable

enumerator **ESP_BT_COD_MAJOR_DEV_TOY**

Toy

enumerator **ESP_BT_COD_MAJOR_DEV_HEALTH**

Health

enumerator **ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED**

Uncategorized: device not specified

enum **esp_bt_gap_discovery_state_t**

Bluetooth Device Discovery state

Values:

enumerator **ESP_BT_GAP_DISCOVERY_STOPPED**

Device discovery stopped

enumerator **ESP_BT_GAP_DISCOVERY_STARTED**

Device discovery started

enum **esp_bt_gap_cb_event_t**

BT GAP callback events.

Values:

enumerator **ESP_BT_GAP_DISC_RES_EVT**

Device discovery result event

enumerator **ESP_BT_GAP_DISC_STATE_CHANGED_EVT**

Discovery state changed event

enumerator **ESP_BT_GAP_RMT_SRVCS_EVT**

Get remote services event

enumerator **ESP_BT_GAP_RMT_SRVC_REC_EVT**

Get remote service record event

enumerator **ESP_BT_GAP_AUTH_CMPL_EVT**

Authentication complete event

enumerator **ESP_BT_GAP_PIN_REQ_EVT**

Legacy Pairing Pin code request

enumerator **ESP_BT_GAP_CFM_REQ_EVT**

Security Simple Pairing User Confirmation request.

enumerator **ESP_BT_GAP_KEY_NOTIF_EVT**

Security Simple Pairing Passkey Notification

enumerator **ESP_BT_GAP_KEY_REQ_EVT**

Security Simple Pairing Passkey request

enumerator **ESP_BT_GAP_READ_RSSI_DELTA_EVT**

Read rssi event

enumerator **ESP_BT_GAP_CONFIG_EIR_DATA_EVT**

Config EIR data event

enumerator **ESP_BT_GAP_SET_AFH_CHANNELS_EVT**

Set AFH channels event

enumerator **ESP_BT_GAP_READ_REMOTE_NAME_EVT**

Read Remote Name event

enumerator **ESP_BT_GAP_MODE_CHG_EVT**

enumerator **ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT**

remove bond device complete event

enumerator **ESP_BT_GAP_QOS_CMPL_EVT**

QOS complete event

enumerator **ESP_BT_GAP_ACL_CONN_CMPL_STAT_EVT**

ACL connection complete status event

enumerator **ESP_BT_GAP_ACL_DISCONN_CMPL_STAT_EVT**

ACL disconnection complete status event

enumerator **ESP_BT_GAP_ACL_PKT_TYPE_CHANGED_EVT**

Set ACL packet types event

enumerator **ESP_BT_GAP_EVT_MAX**

enum **esp_bt_inq_mode_t**

Inquiry Mode

Values:

enumerator **ESP_BT_INQ_MODE_GENERAL_INQUIRY**

General inquiry mode

enumerator **ESP_BT_INQ_MODE_LIMITED_INQUIRY**

Limited inquiry mode

Bluetooth A2DP API

Application Example Check bluetooth/bluedroid/classic_bt folder in ESP-IDF examples, which contains the following application:

- This is a A2DP sink client demo. This demo can be discovered and connected by A2DP source device and receive the audio stream from remote device - bluetooth/bluedroid/classic_bt/a2dp_sink

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_a2dp_api.h

Functions

esp_err_t **esp_a2d_register_callback** (*esp_a2d_cb_t* callback)

Register application callback function to A2DP module. This function should be called only after esp_bluedroid_enable() completes successfully, used by both A2DP source and sink.

Parameters **callback** –**[in]** A2DP event callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_a2d_sink_register_data_callback** (*esp_a2d_sink_data_cb_t* callback)

Register A2DP sink data output function; For now the output is PCM data stream decoded from SBC format. This function should be called only after esp_bluedroid_enable() completes successfully, used only by A2DP sink. The callback is invoked in the context of A2DP sink task whose stack size is configurable through menuconfig.

Parameters **callback** –**[in]** A2DP sink data callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_a2d_sink_init** (void)

Initialize the bluetooth A2DP sink module. This function should be called after esp_bluedroid_enable() completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_INIT_SUCCESS will reported to the APP layer. Note: A2DP can work independently. If you want to use AVRC together, you should initiate AVRC first. This function should be called after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: if the initialization request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_deinit** (void)

De-initialize for A2DP sink module. This function should be called only after esp_bluedroid_enable() completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_DEINIT_SUCCESS will reported to APP layer.

Returns

- ESP_OK: if the deinitialization request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_connect** (*esp_bd_addr_t* remote_bda)

Connect to remote bluetooth A2DP source device. This API must be called after esp_a2d_sink_init() and before esp_a2d_sink_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: connect request is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote A2DP source device. This API must be called after esp_a2d_sink_init() and before esp_a2d_sink_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: disconnect request is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_set_delay_value** (uint16_t delay_value)

Set delay reporting value. The delay value of sink is caused by buffering (including protocol stack and application layer), decoding and rendering. The default delay value is 120ms, if the set value is less than 120ms, the setting will fail. This API must be called after esp_a2d_sink_init() and before esp_a2d_sink_deinit().

Parameters **delay_value** –**[in]** reporting value is in 1/10 millisecond

Returns

- ESP_OK: delay value is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_get_delay_value** (void)

Get delay reporting value. This API must be called after esp_a2d_sink_init() and before esp_a2d_sink_deinit().

Returns

- ESP_OK: if the request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_media_ctrl** (*esp_a2d_media_ctrl_t* ctrl)

Media control commands. This API can be used for both A2DP sink and source and must be called after esp_a2d_sink_init() and before esp_a2d_sink_deinit().

Parameters **ctrl** –**[in]** control commands for A2DP data channel

Returns

- ESP_OK: control command is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_source_init** (void)

Initialize the bluetooth A2DP source module. A2DP can work independently. If you want to use AVRC together, you should initiate AVRC first. This function should be called after esp_bluedroid_enable() completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_INIT_SUCCESS will reported to the APP layer. Note: A2DP can work independently. If you want to use AVRC together, you should initiate AVRC first.

Returns

- ESP_OK: if the initialization request is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_source_deinit** (void)

De-initialize for A2DP source module. This function should be called only after esp_bluedroid_enable() completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_DEINIT_SUCCESS will reported to APP layer.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_source_register_data_callback** (*esp_a2d_source_data_cb_t* callback)

Register A2DP source data input function. For now, the input shoule be PCM data stream. This function should be called only after esp_bluedroid_enable() completes successfully. The callback is invoked in the context of A2DP source task whose stack size is configurable through menuconfig.

Parameters **callback** –**[in]** A2DP source data callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_a2d_source_connect** (*esp_bd_addr_t* remote_bda)

Connect to remote A2DP sink device. This API must be called after esp_a2d_source_init() and before esp_a2d_source_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: connect request is sent to lower layer successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_source_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote A2DP sink device. This API must be called after esp_a2d_source_init() and before esp_a2d_source_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: disconnect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Unions

union **esp_a2d_cb_param_t**

#include <esp_a2dp_api.h> A2DP state callback parameters.

Public Members

struct *esp_a2d_cb_param_t*::*a2d_conn_stat_param* **conn_stat**

A2DP connection status

struct *esp_a2d_cb_param_t*::*a2d_audio_stat_param* **audio_stat**

audio stream playing state

struct *esp_a2d_cb_param_t*::*a2d_audio_cfg_param* **audio_cfg**

media codec configuration information

struct *esp_a2d_cb_param_t*::*media_ctrl_stat_param* **media_ctrl_stat**

status in acknowledgement to media control commands

struct *esp_a2d_cb_param_t*::*a2d_prof_stat_param* **a2d_prof_stat**

status to indicate a2d prof init or deinit

struct *esp_a2d_cb_param_t*::*a2d_psc_cfg_param* **a2d_psc_cfg_stat**

status to indicate protocol service capabilities configured

struct *esp_a2d_cb_param_t*::*a2d_set_stat_param* **a2d_set_delay_value_stat**

A2DP sink set delay report value status

struct *esp_a2d_cb_param_t*::*a2d_get_stat_param* **a2d_get_delay_value_stat**

A2DP sink get delay report value status

struct *esp_a2d_cb_param_t*::*a2d_report_delay_stat_param* **a2d_report_delay_value_stat**

A2DP source received sink report value status

struct **a2d_audio_cfg_param**

#include <*esp_a2dp_api.h*> ESP_A2D_AUDIO_CFG_EVT.

Public Members

esp_bd_addr_t **remote_bda**

remote bluetooth device address

esp_a2d_mcc_t **mcc**

A2DP media codec capability information

struct **a2d_audio_stat_param**

#include <*esp_a2dp_api.h*> ESP_A2D_AUDIO_STATE_EVT.

Public Members

esp_a2d_audio_state_t **state**

one of the values from esp_a2d_audio_state_t

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **a2d_conn_stat_param**

#include <*esp_a2dp_api.h*> ESP_A2D_CONNECTION_STATE_EVT.

Public Members

esp_a2d_connection_state_t **state**

one of values from esp_a2d_connection_state_t

esp_bd_addr_t **remote_bda**

remote bluetooth device address

esp_a2d_disc_rsn_t **disc_rsn**

reason of disconnection for "DISCONNECTED"

struct **a2d_get_stat_param**

#include <*esp_a2dp_api.h*> ESP_A2D_SNK_GET_DELAY_VALUE_EVT.

Public Members

uint16_t **delay_value**
delay report value

struct **a2d_prof_stat_param**
#include <esp_a2dp_api.h> ESP_A2D_PROF_STATE_EVT.

Public Members

esp_a2d_init_state_t **init_state**
a2dp profile state param

struct **a2d_psc_cfg_param**
#include <esp_a2dp_api.h> ESP_A2D_SNK_PSC_CFG_EVT.

Public Members

esp_a2d_psc_t **psc_mask**
protocol service capabilities configured

struct **a2d_report_delay_stat_param**
#include <esp_a2dp_api.h> ESP_A2D_REPORT_SNK_DELAY_VALUE_EVT.

Public Members

uint16_t **delay_value**
delay report value

struct **a2d_set_stat_param**
#include <esp_a2dp_api.h> ESP_A2D_SNK_SET_DELAY_VALUE_EVT.

Public Members

esp_a2d_set_delay_value_state_t **set_state**
a2dp profile state param

uint16_t **delay_value**
delay report value

struct **media_ctrl_stat_param**
#include <esp_a2dp_api.h> ESP_A2D_MEDIA_CTRL_ACK_EVT.

Public Members

esp_a2d_media_ctrl_t **cmd**

media control commands to acknowledge

esp_a2d_media_ctrl_ack_t **status**

acknowledgement to media control commands

Structures

struct **esp_a2d_mcc_t**

A2DP media codec capabilities union.

Public Members

esp_a2d_mct_t **type**

A2DP media codec type

uint8_t **sbc**[ESP_A2D_CIE_LEN_SBC]

SBC codec capabilities

uint8_t **m12**[ESP_A2D_CIE_LEN_M12]

MPEG-1,2 audio codec capabilities

uint8_t **m24**[ESP_A2D_CIE_LEN_M24]

MPEG-2, 4 AAC audio codec capabilities

uint8_t **atrac**[ESP_A2D_CIE_LEN_ATRAC]

ATRAC family codec capabilities

union *esp_a2d_mcc_t*::[anonymous] **cie**

A2DP codec information element

Macros

ESP_A2D_MCT_SBC

Media codec types supported by A2DP.

SBC

ESP_A2D_MCT_M12

MPEG-1, 2 Audio

ESP_A2D_MCT_M24

MPEG-2, 4 AAC

ESP_A2D_MCT_ATRAC

ATRAC family

ESP_A2D_MCT_NON_A2DP

NON-A2DP

ESP_A2D_PSC_DELAY_RPT

Protocol service capabilities. This value is a mask.

Delay Report

ESP_A2D_CIE_LEN_SBC

ESP_A2D_CIE_LEN_M12

ESP_A2D_CIE_LEN_M24

ESP_A2D_CIE_LEN_ATRAC

Type Definitions

typedef uint8_t **esp_a2d_mct_t**

typedef uint16_t **esp_a2d_psc_t**

typedef void (***esp_a2d_cb_t**)(*esp_a2d_cb_event_t* event, *esp_a2d_cb_param_t* *param)

A2DP profile callback function type.

Param event : Event type
Param param : Pointer to callback parameter

typedef void (***esp_a2d_sink_data_cb_t**)(const uint8_t *buf, uint32_t len)

A2DP sink data callback function.

Param buf **[in]** : pointer to the data received from A2DP source device and is PCM format decoded from SBC decoder; buf references to a static memory block and can be overwritten by upcoming data
Param len **[in]** : size(in bytes) in buf

typedef int32_t (***esp_a2d_source_data_cb_t**)(uint8_t *buf, int32_t len)

A2DP source data read callback function.

Param buf **[in]** : buffer to be filled with PCM data stream from higher layer
Param len **[in]** : size(in bytes) of data block to be copied to buf. -1 is an indication to user that data buffer shall be flushed
Return size of bytes read successfully, if the argument len is -1, this value is ignored.

Enumerations

enum **esp_a2d_connection_state_t**

Bluetooth A2DP connection states.

Values:

enumerator **ESP_A2D_CONNECTION_STATE_DISCONNECTED**

connection released

enumerator **ESP_A2D_CONNECTION_STATE_CONNECTING**

connecting remote device

enumerator **ESP_A2D_CONNECTION_STATE_CONNECTED**

connection established

enumerator **ESP_A2D_CONNECTION_STATE_DISCONNECTING**

disconnecting remote device

enum **esp_a2d_disc_rsn_t**

Bluetooth A2DP disconnection reason.

Values:

enumerator **ESP_A2D_DISC_RSN_NORMAL**

Finished disconnection that is initiated by local or remote device

enumerator **ESP_A2D_DISC_RSN_ABNORMAL**

Abnormal disconnection caused by signal loss

enum **esp_a2d_audio_state_t**

Bluetooth A2DP datapath states.

Values:

enumerator **ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND**

audio stream datapath suspended by remote device

enumerator **ESP_A2D_AUDIO_STATE_STOPPED**

audio stream datapath stopped

enumerator **ESP_A2D_AUDIO_STATE_STARTED**

audio stream datapath started

enum **esp_a2d_media_ctrl_ack_t**

A2DP media control command acknowledgement code.

Values:

enumerator **ESP_A2D_MEDIA_CTRL_ACK_SUCCESS**

media control command is acknowledged with success

enumerator **ESP_A2D_MEDIA_CTRL_ACK_FAILURE**

media control command is acknowledged with failure

enumerator **ESP_A2D_MEDIA_CTRL_ACK_BUSY**

media control command is rejected, as previous command is not yet acknowledged

enum **esp_a2d_media_ctrl_t**

A2DP media control commands.

Values:

enumerator **ESP_A2D_MEDIA_CTRL_NONE**

Not for application use, use inside stack only.

enumerator **ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY**

check whether AVDTP is connected, only used in A2DP source

enumerator **ESP_A2D_MEDIA_CTRL_START**

command to set up media transmission channel

enumerator **ESP_A2D_MEDIA_CTRL_STOP**

command to stop media transmission

enumerator **ESP_A2D_MEDIA_CTRL_SUSPEND**

command to suspend media transmission

enum **esp_a2d_init_state_t**

Bluetooth A2DP Initiation states.

Values:

enumerator **ESP_A2D_DEINIT_SUCCESS**

A2DP profile deinit successful event

enumerator **ESP_A2D_INIT_SUCCESS**

A2DP profile deinit successful event

enum **esp_a2d_set_delay_value_state_t**

Bluetooth A2DP set delay report value states.

Values:

enumerator **ESP_A2D_SET_SUCCESS**

A2DP profile set delay report value successful

enumerator **ESP_A2D_SET_INVALID_PARAMS**

A2DP profile set delay report value is invalid parameter

enum **esp_a2d_cb_event_t**

A2DP callback events.

Values:

enumerator **ESP_A2D_CONNECTION_STATE_EVT**

connection state changed event

enumerator **ESP_A2D_AUDIO_STATE_EVT**

audio stream transmission state changed event

enumerator **ESP_A2D_AUDIO_CFG_EVT**

audio codec is configured, only used for A2DP SINK

enumerator **ESP_A2D_MEDIA_CTRL_ACK_EVT**

acknowledge event in response to media control commands

enumerator **ESP_A2D_PROF_STATE_EVT**

indicate a2dp init&deinit complete

enumerator **ESP_A2D_SNK_PSC_CFG_EVT**

protocol service capabilities configured, only used for A2DP SINK

enumerator **ESP_A2D_SNK_SET_DELAY_VALUE_EVT**

indicate a2dp sink set delay report value complete, only used for A2DP SINK

enumerator **ESP_A2D_SNK_GET_DELAY_VALUE_EVT**

indicate a2dp sink get delay report value complete, only used for A2DP SINK

enumerator **ESP_A2D_REPORT_SNK_DELAY_VALUE_EVT**

report delay value, only used for A2DP SRC

BT AVRCP APIs

Overview Bluetooth AVRCP reference APIs.

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_avrc_api.h

Functions

esp_err_t **esp_avrc_ct_register_callback**(*esp_avrc_ct_cb_t* callback)

Register application callbacks to AVRCP module. This function should be called after esp_bluedroid_enable() completes successfully.

Parameters **callback** –**[in]** AVRCP controller callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_init**(void)

Initialize the bluetooth AVRCP controller module, This function should be called after esp_bluedroid_enable() completes successfully. Note: AVRC cannot work independently, AVRC should be used along with A2DP and AVRC should be initialized before A2DP.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_deinit**(void)

De-initialize AVRCP controller module. This function should be called after after esp_bluedroid_enable() completes successfully. Note: AVRC cannot work independently, AVRC should be used along with A2DP and AVRC should be deinitialized before A2DP.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_set_player_value_cmd**(uint8_t tl, uint8_t attr_id, uint8_t value_id)

Send player application settings command to AVRCP target. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters

- **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values
- **attr_id** –**[in]** : player application setting attribute IDs from one of esp_avrc_ps_attr_ids_t
- **value_id** –**[in]** : attribute value defined for the specific player application setting attribute

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_get_rn_capabilities_cmd**(uint8_t tl)

Send GetCapabilities PDU to AVRCP target to retrieve remote device's supported notification event_ids. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_register_notification_cmd**(uint8_t tl, uint8_t event_id, uint32_t event_parameter)

Send register notification command to AVRCP target. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters

- **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values.
- **event_id** –**[in]** : id of events, e.g. ESP_AVRC_RN_PLAY_STATUS_CHANGE, ESP_AVRC_RN_TRACK_CHANGE, etc.
- **event_parameter** –**[in]** : playback interval for ESP_AVRC_RN_PLAY_POS_CHANGED; For other events , value of this parameter is ignored.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_NOT_SUPPORTED: if the event_id is not supported in current implementation
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_set_absolute_volume_cmd**(uint8_t tl, uint8_t volume)

Send set absolute volume command to AVRCP target. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters

- **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values
- **volume** –**[in]** : volume, 0 to 0x7f, means 0% to 100%

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled

- ESP_ERR_NOT_SUPPORTED: if the event_id is not supported in current implementation
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_metadata_cmd**(uint8_t tl, uint8_t attr_mask)

Send metadata command to AVRCP target. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters

- **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values.
- **attr_mask** –**[in]** : mask of attributes, e.g. ESP_AVRC_MD_ATTR_ID_TITLE | ESP_AVRC_MD_ATTR_ID_ARTIST.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_ct_send_passthrough_cmd**(uint8_t tl, uint8_t key_code, uint8_t key_state)

Send passthrough command to AVRCP target. This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Parameters

- **tl** –**[in]** : transaction label, 0 to 15, consecutive commands should use different values.
- **key_code** –**[in]** : passthrough command code, e.g. ESP_AVRC_PT_CMD_PLAY, ESP_AVRC_PT_CMD_STOP, etc.
- **key_state** –**[in]** : passthrough command key state, ESP_AVRC_PT_CMD_STATE_PRESSED or ESP_AVRC_PT_CMD_STATE_RELEASED

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_register_callback**(*esp_avrc_tg_cb_t* callback)

Register application callbacks to AVRCP target module. This function should be called after esp_bluedroid_enable() completes successfully.

Parameters **callback** –**[in]** AVRCP target callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_init**(void)

Initialize the bluetooth AVRCP target module, This function should be called after esp_bluedroid_enable() completes successfully. Note: AVRC cannot work independently, AVRC should be used along with A2DP and AVRC should be initialized before A2DP.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_deinit**(void)

De-initialize AVRCP target module. This function should be called after after esp_bluedroid_enable() completes successfully. Note: AVRC cannot work independently, AVRC should be used along with A2DP and AVRC should be deinitialized before A2DP.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_get_psth_cmd_filter**(*esp_avrc_psth_filter_t* filter, *esp_avrc_psth_bit_mask_t* *cmd_set)

Get the current filter of remote passthrough commands on AVRC target. Filter is given by filter type and bit mask for the passthrough commands. This function should be called after esp_avrc_tg_init(). For filter type ESP_AVRC_PSTH_FILTER_ALLOWED_CMD, the retrieved command set is constant and it covers all of the passthrough commands that can possibly be supported. For filter type ESP_AVRC_PSTH_FILTER_SUPPORT_COMMANDS, the retrieved command set covers the passthrough commands selected to be supported according to current configuration. The configuration can be changed using esp_avrc_tg_set_psth_cmd_filter().

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled or AVRC TG is not initialized
- ESP_ERR_INVALID_ARG: if filter type is invalid or cmd_set is NULL
- ESP_FAIL: otherwise

esp_err_t **esp_avrc_tg_set_psth_cmd_filter**(*esp_avrc_psth_filter_t* filter, const *esp_avrc_psth_bit_mask_t* *cmd_set)

Set the filter of remote passthrough commands on AVRC target. Filter is given by filter type and bit mask for the passthrough commands. This function should be called after esp_avrc_tg_init().

If filter type is ESP_AVRC_PSTH_FILTER_SUPPORT_CMD, the passthrough commands which are set "1" as given in cmd_set will generate ESP_AVRC_CT_PASSTHROUGH_RSP_EVT callback event and are auto-accepted in the protocol stack, other commands are replied with response type "NOT IMPLEMENTED" (8). The set of supported commands should be a subset of allowed command set. The allowed command set can be retrieved using esp_avrc_tg_get_psth_cmd_filter() with filter type "ESP_AVRC_PSTH_FILTER_ALLOWED_CMD" .

Filter type "ESP_AVRC_PSTH_FILTER_ALLOWED_CMD" does not apply to this function.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled
- ESP_ERR_INVALID_ARG: if filter type is invalid or cmd_set is NULL
- ESP_ERR_NOT_SUPPORTED:: if filter type is ESP_AVRC_PSTH_FILTER_ALLOWED_CMD, or cmd_set includes commands that are not allowed

bool **esp_avrc_psth_bit_mask_operation**(*esp_avrc_bit_mask_op_t* op, *esp_avrc_psth_bit_mask_t* *psth, *esp_avrc_pt_cmd_t* cmd)

Operate on the type *esp_avrc_psth_bit_mask_t* with regard to a specific PASSTHROUGH command.

Parameters

- **op** –**[in]** operation requested on the bit mask field
- **psth** –**[in]** pointer to passthrough command bit mask structure
- **cmd** –**[in]** passthrough command code

Returns For operation ESP_AVRC_BIT_MASK_OP_SET or ESP_AVRC_BIT_MASK_OP_CLEAR, return true for a successful operation, otherwise return false. For operation ESP_AVRC_BIT_MASK_OP_TEST, return true if the corresponding bit is set, otherwise false.

esp_err_t **esp_avrc_tg_get_rn_evt_cap**(*esp_avrc_rn_evt_cap_t* cap, *esp_avrc_rn_evt_cap_mask_t* *evt_set)

Get the requested event notification capabilies on local AVRC target. The capability is returned in a bit mask representation in evt_set. This function should be called after esp_avrc_tg_init().

For capability type "ESP_AVRC_RN_CAP_ALLOWED_EVT, the retrieved event set is constant and it covers all of the notifcation events that can possibly be supported with current implementation.

For capability type ESP_AVRC_RN_CAP_SUPPORTED_EVT, the event set covers the notification events selected to be supported under current configuration, The configuration can be changed using

esp_avrc_tg_set_rn_evt_cap().

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled or AVRC TG is not initialized
- ESP_ERR_INVALID_ARG: if cap is invalid or evt_set is NULL
- ESP_FAIL: otherwise

esp_err_t **esp_avrc_tg_set_rn_evt_cap**(const *esp_avrc_rn_evt_cap_mask_t* *evt_set)

Set the event notification capabilities on local AVRCP target. The capability is given in a bit mask representation in evt_set and must be a subset of allowed event IDs with current implementation. This function should be called after esp_avrc_tg_init().

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled
- ESP_ERR_INVALID_ARG: if evt_set is NULL

bool **esp_avrc_rn_evt_bit_mask_operation**(*esp_avrc_bit_mask_op_t* op, *esp_avrc_rn_evt_cap_mask_t* *events, *esp_avrc_rn_event_ids_t* event_id)

Operate on the type *esp_avrc_rn_evt_cap_mask_t* with regard to a specific event.

For operation ESP_AVRC_BIT_MASK_OP_TEST, return true if the corresponding bit is set, otherwise false.

Parameters

- **op** –**[in]** operation requested on the bit mask field
- **events** –**[in]** pointer to event notification capability bit mask structure
- **event_id** –**[in]** notification event code

Returns For operation ESP_AVRC_BIT_MASK_OP_SET or ESP_AVRC_BIT_MASK_OP_CLEAR, return true for a successful operation, otherwise return false.

esp_err_t **esp_avrc_tg_send_rn_rsp**(*esp_avrc_rn_event_ids_t* event_id, *esp_avrc_rn_rsp_t* rsp, *esp_avrc_rn_param_t* *param)

Send RegisterNotification Response to remote AVRCP controller. Local event notification capability can be set using esp_avrc_tg_set_rn_evt_cap(), in a bit mask representation in evt_set. This function should be called after esp_avrc_tg_init().

Parameters

- **event_id** –**[in]** notification event ID that remote AVRCP CT registers
- **rsp** –**[in]** notification response code
- **param** –**[in]** parameters included in the specific notification

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled or AVRC TG is not initialized
- ESP_ERR_INVALID_ARG: if evt_set is NULL

Unions

union **esp_avrc_rn_param_t**

#include <esp_avrc_api.h> AVRCP notification parameters.

Public Members

uint8_t **volume**

response data for ESP_AVRC_RN_VOLUME_CHANGE, ranges 0..127

esp_avrc_playback_stat_t **playback**

response data for ESP_AVRC_RN_PLAY_STATUS_CHANGE

uint8_t **elm_id**[8]

response data for ESP_AVRC_RN_TRACK_CHANGE

uint32_t **play_pos**

response data for ESP_AVRC_RN_PLAY_POS_CHANGED, in millisecond

esp_avrc_batt_stat_t **batt**

response data for ESP_AVRC_RN_BATTERY_STATUS_CHANGE

union **esp_avrc_ct_cb_param_t**

#include <esp_avrc_api.h> AVRC controller callback parameters.

Public Members

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_conn_stat_param* **conn_stat**

AVRC connection status

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_psth_rsp_param* **psth_rsp**

passthrough command response

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_meta_rsp_param* **meta_rsp**

metadata attributes response

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_change_notify_param* **change_ntf**

notifications

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_rmt_feats_param* **rmt_feats**

AVRC features discovered from remote SDP server

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_get_rn_caps_rsp_param* **get_rn_caps_rsp**

get supported event capabilities response from AVRCP target

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_set_volume_rsp_param* **set_volume_rsp**

set absolute volume response event

struct **avrc_ct_change_notify_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_CHANGE_NOTIFY_EVT.

Public Members

uint8_t **event_id**

id of AVRC event notification

esp_avrc_rn_param_t **event_parameter**

event notification parameter

struct **avrc_ct_conn_stat_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_CONNECTION_STATE_EVT.

Public Members

bool **connected**

whether AVRC connection is set up

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **avrc_ct_get_rn_caps_rsp_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT.

Public Members

uint8_t **cap_count**

number of items provided in event or company_id according to cap_id used

esp_avrc_rn_evt_cap_mask_t **evt_set**

supported event_ids represented in bit-mask

struct **avrc_ct_meta_rsp_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_METADATA_RSP_EVT.

Public Members

uint8_t **attr_id**

id of metadata attribute

uint8_t ***attr_text**

attribute itself

int **attr_length**

attribute character length

struct **avrc_ct_psth_rsp_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_PASSTHROUGH_RSP_EVT.

Public Members

uint8_t **tl**

transaction label, 0 to 15

uint8_t **key_code**

passthrough command code

uint8_t **key_state**

0 for PRESSED, 1 for RELEASED

esp_avrc_rsp_t **rsp_code**

response code

struct **avrc_ct_rmt_feats_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_REMOTE_FEATURES_EVT.

Public Members

uint32_t **feat_mask**

AVRC feature mask of remote device

uint16_t **tg_feat_flag**

feature flag of remote device as TG

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **avrc_ct_set_volume_rsp_param**

#include <esp_avrc_api.h> ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT.

Public Members

uint8_t **volume**

the volume which has actually been set, range is 0 to 0x7f, means 0% to 100%

union **esp_avrc_tg_cb_param_t**

#include <esp_avrc_api.h> AVRC target callback parameters.

Public Members

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_conn_stat_param* **conn_stat**

AVRC connection status

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_rmt_feats_param* **rmt_feats**

AVRC features discovered through SDP

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_psth_cmd_param* **psth_cmd**

passthrough command

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_set_abs_vol_param* **set_abs_vol**

set absolute volume command targeted on audio sink

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_reg_ntf_param* **reg_ntf**

register notification

struct *esp_avrc_tg_cb_param_t*::*avrc_tg_set_app_value_param* **set_app_value**

set player application value

struct **avrc_tg_conn_stat_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_CONNECTION_STATE_EVT.

Public Members

bool **connected**

whether AVRC connection is set up

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **avrc_tg_psth_cmd_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_PASSTHROUGH_CMD_EVT.

Public Members

uint8_t **key_code**

passthrough command code

uint8_t **key_state**

0 for PRESSED, 1 for RELEASED

struct **avrc_tg_reg_ntf_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT.

Public Members

uint8_t **event_id**

event id of AVRC RegisterNotification

uint32_t **event_parameter**

event notification parameter

struct **avrc_tg_rmt_feats_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_REMOTE_FEATURES_EVT.

Public Members

uint32_t **feat_mask**

AVRC feature mask of remote device

uint16_t **ct_feat_flag**

feature flag of remote device as CT

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **avrc_tg_set_abs_vol_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT.

Public Members

uint8_t **volume**

volume ranges from 0 to 127

struct **avrc_tg_set_app_value_param**

#include <esp_avrc_api.h> ESP_AVRC_TG_SET_PLAYER_APP_VALUE_EVT.

Public Members

uint8_t **num_val**

attribute num

esp_avrc_set_app_value_param_t ***p_vals**

point to the id and value of player application attribute

Structures

struct **esp_avrc_psth_bit_mask_t**

AVRC passthrough command bit mask.

Public Members

uint16_t **bits**[8]

bit mask representation of PASSTHROUGH commands

struct **esp_avrc_rn_evt_cap_mask_t**

AVRC target notification event capability bit mask.

Public Members

uint16_t **bits**

bit mask representation of PASSTHROUGH commands

struct **esp_avrc_set_app_value_param_t**

AVRCP set app value parameters.

Public Members

uint8_t **attr_id**

player application attribute id

uint8_t **attr_val**

player application attribute value

Macros

ESP_AVRC_TRANS_LABEL_MAX

max transaction label

Type Definitions

typedef void (***esp_avrc_ct_cb_t**)(*esp_avrc_ct_cb_event_t* event, *esp_avrc_ct_cb_param_t* *param)

AVRCP controller callback function type.

Param event : Event type
Param param : Pointer to callback parameter union

typedef void (***esp_avrc_tg_cb_t**)(*esp_avrc_tg_cb_event_t* event, *esp_avrc_tg_cb_param_t* *param)

AVRCP target callback function type.

Param event : Event type
Param param : Pointer to callback parameter union

Enumerations

enum **esp_avrc_features_t**

AVRC feature bit mask.

Values:

enumerator **ESP_AVRC_FEAT_RCTG**

remote control target

enumerator **ESP_AVRC_FEAT_RCCT**

remote control controller

enumerator **ESP_AVRC_FEAT_VENDOR**

remote control vendor dependent commands

enumerator **ESP_AVRC_FEAT_BROWSE**

use browsing channel

enumerator **ESP_AVRC_FEAT_META_DATA**

remote control metadata transfer command/response

enumerator **ESP_AVRC_FEAT_ADV_CTRL**

remote control advanced control command/response

enum **esp_avrc_feature_flag_t**

AVRC supported features flag retrieved in SDP record.

Values:

enumerator **ESP_AVRC_FEAT_FLAG_CAT1**

category 1

enumerator **ESP_AVRC_FEAT_FLAG_CAT2**

category 2

enumerator **ESP_AVRC_FEAT_FLAG_CAT3**

category 3

enumerator **ESP_AVRC_FEAT_FLAG_CAT4**

category 4

enumerator **ESP_AVRC_FEAT_FLAG_BROWSING**

browsing

enumerator **ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE_PROP**

Cover Art GetImageProperties

enumerator **ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE**

Cover Art GetImage

enumerator **ESP_AVRC_FEAT_FLAG_COVER_ART_GET_LINKED_THUMBNAIL**

Cover Art GetLinkedThumbnail

enum **esp_avrc_pt_cmd_t**

AVRC passthrough command code.

Values:

enumerator **ESP_AVRC_PT_CMD_SELECT**

select

enumerator **ESP_AVRC_PT_CMD_UP**

up

enumerator **ESP_AVRC_PT_CMD_DOWN**

down

enumerator **ESP_AVRC_PT_CMD_LEFT**

left

enumerator **ESP_AVRC_PT_CMD_RIGHT**
right

enumerator **ESP_AVRC_PT_CMD_RIGHT_UP**
right-up

enumerator **ESP_AVRC_PT_CMD_RIGHT_DOWN**
right-down

enumerator **ESP_AVRC_PT_CMD_LEFT_UP**
left-up

enumerator **ESP_AVRC_PT_CMD_LEFT_DOWN**
left-down

enumerator **ESP_AVRC_PT_CMD_ROOT_MENU**
root menu

enumerator **ESP_AVRC_PT_CMD_SETUP_MENU**
setup menu

enumerator **ESP_AVRC_PT_CMD_CONT_MENU**
contents menu

enumerator **ESP_AVRC_PT_CMD_FAV_MENU**
favorite menu

enumerator **ESP_AVRC_PT_CMD_EXIT**
exit

enumerator **ESP_AVRC_PT_CMD_0**
0

enumerator **ESP_AVRC_PT_CMD_1**
1

enumerator **ESP_AVRC_PT_CMD_2**
2

enumerator **ESP_AVRC_PT_CMD_3**
3

enumerator **ESP_AVRC_PT_CMD_4**
4

enumerator **ESP_AVRC_PT_CMD_5**
5

enumerator **ESP_AVRC_PT_CMD_6**
6

enumerator **ESP_AVRC_PT_CMD_7**
7

enumerator **ESP_AVRC_PT_CMD_8**
8

enumerator **ESP_AVRC_PT_CMD_9**
9

enumerator **ESP_AVRC_PT_CMD_DOT**
dot

enumerator **ESP_AVRC_PT_CMD_ENTER**
enter

enumerator **ESP_AVRC_PT_CMD_CLEAR**
clear

enumerator **ESP_AVRC_PT_CMD_CHAN_UP**
channel up

enumerator **ESP_AVRC_PT_CMD_CHAN_DOWN**
channel down

enumerator **ESP_AVRC_PT_CMD_PREV_CHAN**
previous channel

enumerator **ESP_AVRC_PT_CMD_SOUND_SEL**
sound select

enumerator **ESP_AVRC_PT_CMD_INPUT_SEL**
input select

enumerator **ESP_AVRC_PT_CMD_DISP_INFO**
display information

enumerator **ESP_AVRC_PT_CMD_HELP**
help

enumerator **ESP_AVRC_PT_CMD_PAGE_UP**
page up

enumerator **ESP_AVRC_PT_CMD_PAGE_DOWN**
page down

enumerator **ESP_AVRC_PT_CMD_POWER**
power

enumerator **ESP_AVRC_PT_CMD_VOL_UP**
volume up

enumerator **ESP_AVRC_PT_CMD_VOL_DOWN**
volume down

enumerator **ESP_AVRC_PT_CMD_MUTE**
mute

enumerator **ESP_AVRC_PT_CMD_PLAY**
play

enumerator **ESP_AVRC_PT_CMD_STOP**
stop

enumerator **ESP_AVRC_PT_CMD_PAUSE**
pause

enumerator **ESP_AVRC_PT_CMD_RECORD**
record

enumerator **ESP_AVRC_PT_CMD_REWIND**
rewind

enumerator **ESP_AVRC_PT_CMD_FAST_FORWARD**
fast forward

enumerator **ESP_AVRC_PT_CMD_EJECT**
eject

enumerator **ESP_AVRC_PT_CMD_FORWARD**
forward

enumerator **ESP_AVRC_PT_CMD_BACKWARD**
backward

enumerator **ESP_AVRC_PT_CMD_ANGLE**
angle

enumerator **ESP_AVRC_PT_CMD_SUBPICT**
subpicture

enumerator **ESP_AVRC_PT_CMD_F1**
F1

enumerator **ESP_AVRC_PT_CMD_F2**

F2

enumerator **ESP_AVRC_PT_CMD_F3**

F3

enumerator **ESP_AVRC_PT_CMD_F4**

F4

enumerator **ESP_AVRC_PT_CMD_F5**

F5

enumerator **ESP_AVRC_PT_CMD_VENDOR**

vendor unique

enum **esp_avrc_psth_filter_t**

AVRC passthrough command filter.

Values:

enumerator **ESP_AVRC_PSTH_FILTER_ALLOWED_CMD**

all of the PASSTHROUGH commands that can possibly be used, immutable

enumerator **ESP_AVRC_PSTH_FILTER_SUPPORTED_CMD**

PASSTHROUGH commands selectively supported according to the current configuration

enumerator **ESP_AVRC_PSTH_FILTER_SUPPORT_MAX**

enum **esp_avrc_bit_mask_op_t**

Values:

enumerator **ESP_AVRC_BIT_MASK_OP_TEST**

operation code to test a specific bit

enumerator **ESP_AVRC_BIT_MASK_OP_SET**

operation code to set a specific bit

enumerator **ESP_AVRC_BIT_MASK_OP_CLEAR**

operation code to clear a specific bit

enum **esp_avrc_pt_cmd_state_t**

AVRC passthrough command state.

Values:

enumerator **ESP_AVRC_PT_CMD_STATE_PRESSED**

key pressed

enumerator **ESP_AVRC_PT_CMD_STATE_RELEASED**

key released

enum **esp_avrc_ct_cb_event_t**

AVRC Controller callback events.

Values:

enumerator **ESP_AVRC_CT_CONNECTION_STATE_EVT**

connection state changed event

enumerator **ESP_AVRC_CT_PASSTHROUGH_RSP_EVT**

passthrough response event

enumerator **ESP_AVRC_CT_METADATA_RSP_EVT**

metadata response event

enumerator **ESP_AVRC_CT_PLAY_STATUS_RSP_EVT**

play status response event

enumerator **ESP_AVRC_CT_CHANGE_NOTIFY_EVT**

notification event

enumerator **ESP_AVRC_CT_REMOTE_FEATURES_EVT**

feature of remote device indication event

enumerator **ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT**

supported notification events capability of peer device

enumerator **ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT**

set absolute volume response event

enum **esp_avrc_tg_cb_event_t**

AVRC Target callback events.

Values:

enumerator **ESP_AVRC_TG_CONNECTION_STATE_EVT**

connection state changed event

enumerator **ESP_AVRC_TG_REMOTE_FEATURES_EVT**

feature of remote device indication event

enumerator **ESP_AVRC_TG_PASSTHROUGH_CMD_EVT**

passthrough command event

enumerator **ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT**

set absolute volume command from remote device

enumerator **ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT**

register notification event

enumerator **ESP_AVRC_TG_SET_PLAYER_APP_VALUE_EVT**

set application attribute value, attribute refer to esp_avrc_ps_attr_ids_t

enum **esp_avrc_md_attr_mask_t**

AVRC metadata attribute mask.

Values:

enumerator **ESP_AVRC_MD_ATTR_TITLE**

title of the playing track

enumerator **ESP_AVRC_MD_ATTR_ARTIST**

track artist

enumerator **ESP_AVRC_MD_ATTR_ALBUM**

album name

enumerator **ESP_AVRC_MD_ATTR_TRACK_NUM**

track position on the album

enumerator **ESP_AVRC_MD_ATTR_NUM_TRACKS**

number of tracks on the album

enumerator **ESP_AVRC_MD_ATTR_GENRE**

track genre

enumerator **ESP_AVRC_MD_ATTR_PLAYING_TIME**

total album playing time in miliseconds

enum **esp_avrc_rn_event_ids_t**

AVRC event notification ids.

Values:

enumerator **ESP_AVRC_RN_PLAY_STATUS_CHANGE**

track status change, eg. from playing to paused

enumerator **ESP_AVRC_RN_TRACK_CHANGE**

new track is loaded

enumerator **ESP_AVRC_RN_TRACK_REACHED_END**

current track reached end

enumerator **ESP_AVRC_RN_TRACK_REACHED_START**

current track reached start position

enumerator **ESP_AVRC_RN_PLAY_POS_CHANGED**

track playing position changed

enumerator **ESP_AVRC_RN_BATTERY_STATUS_CHANGE**

battery status changed

enumerator **ESP_AVRC_RN_SYSTEM_STATUS_CHANGE**

system status changed

enumerator **ESP_AVRC_RN_APP_SETTING_CHANGE**

application settings changed

enumerator **ESP_AVRC_RN_NOW_PLAYING_CHANGE**

now playing content changed

enumerator **ESP_AVRC_RN_AVAILABLE_PLAYERS_CHANGE**

available players changed

enumerator **ESP_AVRC_RN_ADDRESSED_PLAYER_CHANGE**

the addressed player changed

enumerator **ESP_AVRC_RN_UIDS_CHANGE**

UIDs changed

enumerator **ESP_AVRC_RN_VOLUME_CHANGE**

volume changed locally on TG

enumerator **ESP_AVRC_RN_MAX_EVT**

enum **esp_avrc_rn_evt_cap_t**

AVRC target notification event notification capability.

Values:

enumerator **ESP_AVRC_RN_CAP_ALLOWED_EVT**

all of the notification events that can possibly be supported, immutable

enumerator **ESP_AVRC_RN_CAP_SUPPORTED_EVT**

notification events selectively supported according to the current configuration

enumerator **ESP_AVRC_RN_CAP_MAX**

enum **esp_avrc_rn_rsp_t**

AVRC notification response type.

Values:

enumerator **ESP_AVRC_RN_RSP_INTERIM**

initial response to RegisterNotification, should be sent T_mtp(1000ms) from receiving the command

enumerator **ESP_AVRC_RN_RSP_CHANGED**

final response to RegisterNotification command

enum **esp_avrc_ps_attr_ids_t**

AVRC player setting ids.

Values:

enumerator **ESP_AVRC_PS_EQUALIZER**

equalizer, on or off

enumerator **ESP_AVRC_PS_REPEAT_MODE**

repeat mode

enumerator **ESP_AVRC_PS_SHUFFLE_MODE**

shuffle mode

enumerator **ESP_AVRC_PS_SCAN_MODE**

scan mode on or off

enumerator **ESP_AVRC_PS_MAX_ATTR**

enum **esp_avrc_ps_eq_value_ids_t**

AVRC equalizer modes.

Values:

enumerator **ESP_AVRC_PS_EQUALIZER_OFF**

equalizer OFF

enumerator **ESP_AVRC_PS_EQUALIZER_ON**

equalizer ON

enum **esp_avrc_ps_rpt_value_ids_t**

AVRC repeat modes.

Values:

enumerator **ESP_AVRC_PS_REPEAT_OFF**

repeat mode off

enumerator **ESP_AVRC_PS_REPEAT_SINGLE**

single track repeat

enumerator **ESP_AVRC_PS_REPEAT_GROUP**

group repeat

enum **esp_avrc_ps_shf_value_ids_t**

AVRC shuffle modes.

Values:

enumerator **ESP_AVRC_PS_SHUFFLE_OFF**

enumerator **ESP_AVRC_PS_SHUFFLE_ALL**

enumerator **ESP_AVRC_PS_SHUFFLE_GROUP**

enum **esp_avrc_ps_scn_value_ids_t**

AVRC scan modes.

Values:

enumerator **ESP_AVRC_PS_SCAN_OFF**

scan off

enumerator **ESP_AVRC_PS_SCAN_ALL**

all tracks scan

enumerator **ESP_AVRC_PS_SCAN_GROUP**

group scan

enum **esp_avrc_rsp_t**

AVCTP response codes.

Values:

enumerator **ESP_AVRC_RSP_NOT_IMPL**

not implemented

enumerator **ESP_AVRC_RSP_ACCEPT**

accept

enumerator **ESP_AVRC_RSP_REJECT**

reject

enumerator **ESP_AVRC_RSP_IN_TRANS**

in transition

enumerator **ESP_AVRC_RSP_IMPL_STBL**

implemented/stable

enumerator **ESP_AVRC_RSP_CHANGED**

changed

enumerator **ESP_AVRC_RSP_INTERIM**

interim

enum **esp_avrc_batt_stat_t**

AVRCP battery status.

Values:

enumerator **ESP_AVRC_BATT_NORMAL**

normal state

enumerator **ESP_AVRC_BATT_WARNING**

unable to operate soon

enumerator **ESP_AVRC_BATT_CRITICAL**

cannot operate any more

enumerator **ESP_AVRC_BATT_EXTERNAL**

plugged to external power supply

enumerator **ESP_AVRC_BATT_FULL_CHARGE**

when completely charged from external power supply

enum **esp_avrc_playback_stat_t**

AVRCP current status of playback.

Values:

enumerator **ESP_AVRC_PLAYBACK_STOPPED**

stopped

enumerator **ESP_AVRC_PLAYBACK_PLAYING**

playing

enumerator **ESP_AVRC_PLAYBACK_PAUSED**

paused

enumerator **ESP_AVRC_PLAYBACK_FWD_SEEK**

forward seek

enumerator **ESP_AVRC_PLAYBACK_REV_SEEK**

reverse seek

enumerator **ESP_AVRC_PLAYBACK_ERROR**

error

SPP API

Application Example Check bluetooth/bluedroid/classic_bt folder in ESP-IDF examples, which contains the following application:

- This is a SPP demo. This demo can discover the service, connect, send and recive SPP data bluetooth/bluedroid/classic_bt/bt_spp_acceptor, bluetooth/bluedroid/classic_bt/bt_spp_initiator

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_spp_api.h

Functions

esp_err_t **esp_spp_register_callback** (*esp_spp_cb_t* callback)

This function is called to init callbacks with SPP module.

Parameters **callback** –**[in]** pointer to the init callback function.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_init** (*esp_spp_mode_t* mode)

This function is called to init SPP module. When the operation is completed, the callback function will be called with ESP_SPP_INIT_EVT. This function should be called after esp_bluedroid_enable() completes successfully.

Parameters **mode** –**[in]** Choose the mode of SPP, ESP_SPP_MODE_CB or ESP_SPP_MODE_VFS.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_enhanced_init** (const *esp_spp_cfg_t* *cfg)

This function is called to init SPP module. When the operation is completed, the callback function will be called with ESP_SPP_INIT_EVT. This function should be called after esp_bluedroid_enable() completes successfully.

Note: The member variable enable_l2cap_etrm in *esp_spp_cfg_t* can affect all L2CAP channel configurations of the upper layer RFCOMM protocol.

Parameters **cfg** –**[in]** SPP configuration.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_deinit** (void)

This function is called to uninit SPP module. The operation will close all active SPP connection first, then the callback function will be called with ESP_SPP_CLOSE_EVT, and the number of ESP_SPP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback function will be called with ESP_SPP_UNINIT_EVT. This function should be called after esp_spp_init()/esp_spp_enhanced_init() completes successfully.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_start_discovery** (*esp_bd_addr_t* bd_addr)

This function is called to performs service discovery for the services provided by the given peer device. When the operation is completed, the callback function will be called with ESP_SPP_DISCOVERY_COMP_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_connect** (*esp_spp_sec_t* sec_mask, *esp_spp_role_t* role, uint8_t remote_scn, *esp_bd_addr_t* peer_bd_addr)

This function makes an SPP connection to a remote BD Address. When the connection is initiated or

failed to initiate, the callback is called with ESP_SPP_CL_INIT_EVT. When the connection is established or failed, the callback is called with ESP_SPP_OPEN_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Parameters

- **sec_mask** –**[in]** Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- **role** –**[in]** Master or slave.
- **remote_scn** –**[in]** Remote device bluetooth device SCN.
- **peer_bd_addr** –**[in]** Remote device bluetooth device address.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_disconnect**(uint32_t handle)

This function closes an SPP connection. When the operation is completed, the callback function will be called with ESP_SPP_CLOSE_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Parameters **handle** –**[in]** The connection handle.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_start_srv**(*esp_spp_sec_t* sec_mask, *esp_spp_role_t* role, uint8_t local_scn, const char *name)

This function create a SPP server and starts listening for an SPP connection request from a remote Bluetooth device. When the server is started successfully, the callback is called with ESP_SPP_START_EVT. When the connection is established, the callback is called with ESP_SPP_SRV_OPEN_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Parameters

- **sec_mask** –**[in]** Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- **role** –**[in]** Master or slave.
- **local_scn** –**[in]** The specific channel you want to get. If channel is 0, means get any channel.
- **name** –**[in]** Server’ s name.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_stop_srv**(void)

This function stops all SPP servers. The operation will close all active SPP connection first, then the callback function will be called with ESP_SPP_CLOSE_EVT, and the number of ESP_SPP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_SPP_SRV_STOP_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_stop_srv_scn**(uint8_t scn)

This function stops a specific SPP server. The operation will close all active SPP connection first on the specific SPP server, then the callback function will be called with ESP_SPP_CLOSE_EVT, and the number of ESP_SPP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_SPP_SRV_STOP_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Parameters **scn** –**[in]** Server channel number.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_write**(uint32_t handle, int len, uint8_t *p_data)

This function is used to write data, only for ESP_SPP_MODE_CB. When this function need to be called repeatedly, it is strongly recommended to call this function again after the previous event ESP_SPP_WRITE_EVT is received and the parameter 'cong' is equal to false. If the previous event ESP_SPP_WRITE_EVT with parameter 'cong' is equal to true, the function can only be called again when the event ESP_SPP_CONG_EVT with parameter 'cong' equal to false is received. This function must be called after an connection between initiator and acceptor has been established.

Parameters

- **handle** –**[in]** The connection handle.
- **len** –**[in]** The length of the data written.
- **p_data** –**[in]** The data written.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_vfs_register**(void)

This function is used to register VFS. For now, SPP only supports write, read and close. When the operation is completed, the callback function will be called with ESP_SPP_VFS_REGISTER_EVT. This function must be called after esp_spp_init()/esp_spp_enhanced_init() successful and before esp_spp_deinit().

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_vfs_unregister**(void)

This function is used to unregister VFS. When the operation is completed, the callback function will be called with ESP_SPP_VFS_UNREGISTER_EVT. This function must be called after esp_spp_vfs_register() successful and before esp_spp_deinit().

Returns

- ESP_OK: success
- other: failed

Unions

union **esp_spp_cb_param_t**

#include <esp_spp_api.h> SPP callback parameters union.

Public Members

struct *esp_spp_cb_param_t*::*spp_init_evt_param* **init**

SPP callback param of SPP_INIT_EVT

struct *esp_spp_cb_param_t*::*spp_uninit_evt_param* **uninit**

SPP callback param of SPP_UNINIT_EVT

struct *esp_spp_cb_param_t*::*spp_discovery_comp_evt_param* **disc_comp**

SPP callback param of SPP_DISCOVERY_COMP_EVT

struct *esp_spp_cb_param_t*::*spp_open_evt_param* **open**

SPP callback param of ESP_SPP_OPEN_EVT

struct *esp_spp_cb_param_t*::*spp_srv_open_evt_param* **srv_open**

SPP callback param of ESP_SPP_SRV_OPEN_EVT

struct *esp_spp_cb_param_t*::*spp_close_evt_param* **close**

SPP callback param of ESP_SPP_CLOSE_EVT

struct *esp_spp_cb_param_t*::*spp_start_evt_param* **start**

SPP callback param of ESP_SPP_START_EVT

struct *esp_spp_cb_param_t*::*spp_srv_stop_evt_param* **srv_stop**

SPP callback param of ESP_SPP_SRV_STOP_EVT

struct *esp_spp_cb_param_t*::*spp_cl_init_evt_param* **cl_init**

SPP callback param of ESP_SPP_CL_INIT_EVT

struct *esp_spp_cb_param_t*::*spp_write_evt_param* **write**

SPP callback param of ESP_SPP_WRITE_EVT

struct *esp_spp_cb_param_t*::*spp_data_ind_evt_param* **data_ind**

SPP callback param of ESP_SPP_DATA_IND_EVT

struct *esp_spp_cb_param_t*::*spp_cong_evt_param* **cong**

SPP callback param of ESP_SPP_CONG_EVT

struct *esp_spp_cb_param_t*::*spp_vfs_register_evt_param* **vfs_register**

SPP callback param of ESP_SPP_VFS_REGISTER_EVT

struct *esp_spp_cb_param_t*::*spp_vfs_unregister_evt_param* **vfs_unregister**

SPP callback param of ESP_SPP_VFS_UNREGISTER_EVT

struct **spp_cl_init_evt_param**

#include <esp_spp_api.h> ESP_SPP_CL_INIT_EVT.

Public Members

esp_spp_status_t **status**

status

uint32_t **handle**

The connection handle

uint8_t **sec_id**

security ID used by this server

bool **use_co**

TRUE to use co_rfc_data

struct **spp_close_evt_param**

#include <esp_spp_api.h> ESP_SPP_CLOSE_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **port_status**
PORT status

uint32_t **handle**
The connection handle

bool **async**
FALSE, if local initiates disconnect

struct **spp_cong_evt_param**
#include <esp_spp_api.h> ESP_SPP_CONG_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

bool **cong**
TRUE, congested. FALSE, uncongested

struct **spp_data_ind_evt_param**
#include <esp_spp_api.h> ESP_SPP_DATA_IND_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

uint16_t **len**
The length of data

uint8_t ***data**
The data received

struct **spp_discovery_comp_evt_param**
#include <esp_spp_api.h> SPP_DISCOVERY_COMP_EVT.

Public Members

esp_spp_status_t **status**
status

uint8_t **scn_num**
The num of scn_num

uint8_t **scn**[ESP_SPP_MAX_SCN]
channel #

const char ***service_name**[ESP_SPP_MAX_SCN]
service_name

struct **spp_init_evt_param**
#include <esp_spp_api.h> SPP_INIT_EVT.

Public Members

esp_spp_status_t **status**
status

struct **spp_open_evt_param**
#include <esp_spp_api.h> ESP_SPP_OPEN_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

int **fd**
The file descriptor only for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**
The peer address

struct **spp_srv_open_evt_param**
#include <esp_spp_api.h> ESP_SPP_SRV_OPEN_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**

The connection handle

uint32_t **new_listen_handle**

The new listen handle

int **fd**

The file descriptor only for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**

The peer address

struct **spp_srv_stop_evt_param**

#include <esp_spp_api.h> ESP_SPP_SRV_STOP_EVT.

Public Members

esp_spp_status_t **status**

status

uint8_t **scn**

Server channel number

struct **spp_start_evt_param**

#include <esp_spp_api.h> ESP_SPP_START_EVT.

Public Members

esp_spp_status_t **status**

status

uint32_t **handle**

The connection handle

uint8_t **sec_id**

security ID used by this server

uint8_t **scn**

Server channel number

bool **use_co**

TRUE to use co_rfc_data

struct **spp_uninit_evt_param**

#include <esp_spp_api.h> SPP_UNINIT_EVT.

Public Members

esp_spp_status_t **status**

status

struct **spp_vfs_register_evt_param**

#include <esp_spp_api.h> ESP_SPP_VFS_REGISTER_EVT.

Public Members

esp_spp_status_t **status**

status

struct **spp_vfs_unregister_evt_param**

#include <esp_spp_api.h> ESP_SPP_VFS_UNREGISTER_EVT.

Public Members

esp_spp_status_t **status**

status

struct **spp_write_evt_param**

#include <esp_spp_api.h> ESP_SPP_WRITE_EVT.

Public Members

esp_spp_status_t **status**

status

uint32_t **handle**

The connection handle

int **len**

The length of the data written.

bool **cong**

congestion status

Structures

struct **esp_spp_cfg_t**

SPP configuration parameters.

Public Members

esp_spp_mode_t **mode**

Choose the mode of SPP, ESP_SPP_MODE_CB or ESP_SPP_MODE_VFS.

bool **enable_l2cap_ertm**

Enable/disable Logical Link Control and Adaptation Layer Protocol enhanced retransmission mode.

uint16_t **tx_buffer_size**

Tx buffer size for a new SPP channel. A smaller setting can save memory, but may incur a decrease in throughput. Only for ESP_SPP_MODE_VFS mode.

Macros

ESP_SPP_MAX_MTU

SPP max MTU

ESP_SPP_MAX_SCN

SPP max SCN

ESP_SPP_MIN_TX_BUFFER_SIZE

SPP min tx buffer

ESP_SPP_MAX_TX_BUFFER_SIZE

SPP max tx buffer size

BT_SPP_DEFAULT_CONFIG()

SPP default configuration.

ESP_SPP_SEC_NONE

No security. relate to BTA_SEC_NONE in bta/bta_api.h

ESP_SPP_SEC_AUTHORIZE

Authorization required (only needed for out going connection) relate to BTA_SEC_AUTHORIZE in bta/bta_api.h

ESP_SPP_SEC_AUTHENTICATE

Authentication required. relate to BTA_SEC_AUTHENTICATE in bta/bta_api.h

ESP_SPP_SEC_ENCRYPT

Encryption required. relate to BTA_SEC_ENCRYPT in bta/bta_api.h

ESP_SPP_SEC_MODE4_LEVEL4

Mode 4 level 4 service, i.e. incoming/outgoing MITM and P-256 encryption relate to BTA_SEC_MODE4_LEVEL4 in bta/bta_api.h

ESP_SPP_SEC_MITM

Man-In-The_Middle protection relate to BTA_SEC_MITM in bta/bta_api.h

ESP_SPP_SEC_IN_16_DIGITS

Min 16 digit for pin code relate to BTA_SEC_IN_16_DIGITS in bta/bta_api.h

Type Definitions

typedef uint16_t **esp_spp_sec_t**

typedef void (***esp_spp_cb_t**)(*esp_spp_cb_event_t* event, *esp_spp_cb_param_t* *param)

SPP callback function type. When handle ESP_SPP_DATA_IND_EVT, it is strongly recommended to cache incoming data, and process them in other lower priority application task rather than in this callback directly.

Param event Event type
Param param Point to callback parameter, currently is union type

Enumerations

enum **esp_spp_status_t**

Values:

enumerator **ESP_SPP_SUCCESS**

Successful operation.

enumerator **ESP_SPP_FAILURE**

Generic failure.

enumerator **ESP_SPP_BUSY**

Temporarily can not handle this request.

enumerator **ESP_SPP_NO_DATA**

No data

enumerator **ESP_SPP_NO_RESOURCE**

No more resource

enumerator **ESP_SPP_NEED_INIT**

SPP module shall init first

enumerator **ESP_SPP_NEED_DEINIT**

SPP module shall deinit first

enumerator **ESP_SPP_NO_CONNECTION**

Connection may have been closed

enumerator **ESP_SPP_NO_SERVER**

No SPP server

enum **esp_spp_role_t**

Values:

enumerator **ESP_SPP_ROLE_MASTER**

Role: master

enumerator **ESP_SPP_ROLE_SLAVE**

Role: slave

enum **esp_spp_mode_t**

Values:

enumerator **ESP_SPP_MODE_CB**

When data is coming, a callback will come with data

enumerator **ESP_SPP_MODE_VFS**

Use VFS to write/read data

enum **esp_spp_cb_event_t**

SPP callback function events.

Values:

enumerator **ESP_SPP_INIT_EVT**

When SPP is initialized, the event comes

enumerator **ESP_SPP_UNINIT_EVT**

When SPP is deinitialized, the event comes

enumerator **ESP_SPP_DISCOVERY_COMP_EVT**

When SDP discovery complete, the event comes

enumerator **ESP_SPP_OPEN_EVT**

When SPP Client connection open, the event comes

enumerator **ESP_SPP_CLOSE_EVT**

When SPP connection closed, the event comes

enumerator **ESP_SPP_START_EVT**

When SPP server started, the event comes

enumerator **ESP_SPP_CL_INIT_EVT**

When SPP client initiated a connection, the event comes

enumerator **ESP_SPP_DATA_IND_EVT**

When SPP connection received data, the event comes, only for ESP_SPP_MODE_CB

enumerator **ESP_SPP_CONG_EVT**

When SPP connection congestion status changed, the event comes, only for ESP_SPP_MODE_CB

enumerator **ESP_SPP_WRITE_EVT**

When SPP write operation completes, the event comes, only for ESP_SPP_MODE_CB

enumerator **ESP_SPP_SRV_OPEN_EVT**

When SPP Server connection open, the event comes

enumerator **ESP_SPP_SRV_STOP_EVT**

When SPP server stopped, the event comes

enumerator **ESP_SPP_VFS_REGISTER_EVT**

When SPP VFS register, the event comes

enumerator **ESP_SPP_VFS_UNREGISTER_EVT**

When SPP VFS unregister, the event comes

HFP DEFINES

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_hf_defs.h

Enumerations

enum **esp_hf_in_band_ring_state_t**

in-band ring tone state

Values:

enumerator **ESP_HF_IN_BAND_RINGTONE_NOT_PROVIDED**

enumerator **ESP_HF_IN_BAND_RINGTONE_PROVIDED**

enum **esp_hf_vr_state_t**

voice recognition state

Values:

enumerator **ESP_HF_VR_STATE_DISABLED**

voice recognition disabled

enumerator **ESP_HF_VR_STATE_ENABLED**

voice recognition enabled

enum **esp_hf_volume_control_target_t**

Bluetooth HFP audio volume control target.

Values:

enumerator **ESP_HF_VOLUME_CONTROL_TARGET_SPK**

speaker

enumerator **ESP_HF_VOLUME_CONTROL_TARGET_MIC**

microphone

enum **esp_hf_audio_state_t**

Bluetooth HFP audio connection status.

Values:

enumerator **ESP_HF_AUDIO_STATE_DISCONNECTED**
audio connection released

enumerator **ESP_HF_AUDIO_STATE_CONNECTING**
audio connection has been initiated

enumerator **ESP_HF_AUDIO_STATE_CONNECTED**
audio connection is established

enumerator **ESP_HF_AUDIO_STATE_CONNECTED_MSBC**
mSBC audio connection is established

enum **esp_hf_volume_type_t**
Values:

enumerator **ESP_HF_VOLUME_TYPE_SPK**

enumerator **ESP_HF_VOLUME_TYPE_MIC**

enum **esp_hf_network_state_t**
+CIND network service availability status
Values:

enumerator **ESP_HF_NETWORK_STATE_NOT_AVAILABLE**

enumerator **ESP_HF_NETWORK_STATE_AVAILABLE**

enum **esp_hf_ciev_report_type_t**
+CIEV report type
Values:

enumerator **ESP_HF_IND_TYPE_CALL**
position of call indicator

enumerator **ESP_HF_IND_TYPE_CALLSETUP**
position of callsetup indicator

enumerator **ESP_HF_IND_TYPE_SERVICE**
position of service indicator

enumerator **ESP_HF_IND_TYPE_SIGNAL**
position of signal strength indicator, range: 0-5

enumerator **ESP_HF_IND_TYPE_ROAM**
position of roaming indicator

enumerator **ESP_HF_IND_TYPE_BATTCHG**
position of battery charge indicator, range: 0-5

enumerator **ESP_HF_IND_TYPE_CALLHELD**

position of callheld indicator

enum **esp_hf_service_type_t**

+CIEV Service type

Values:

enumerator **ESP_HF_SERVICE_TYPE_HOME**

enumerator **ESP_HF_SERVICE_TYPE_ROAMING**

enum **esp_hf_call_status_t**

+CIND call status indicator values

Values:

enumerator **ESP_HF_CALL_STATUS_NO_CALLS**

no call in progress

enumerator **ESP_HF_CALL_STATUS_CALL_IN_PROGRESS**

call is present(active or held)

enum **esp_hf_call_setup_status_t**

+CIND call setup status indicator values

Values:

enumerator **ESP_HF_CALL_SETUP_STATUS_IDLE**

no call setup in progress

enumerator **ESP_HF_CALL_SETUP_STATUS_INCOMING**

incoming call setup in progress

enumerator **ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING**

outgoing call setup in dialing state

enumerator **ESP_HF_CALL_SETUP_STATUS_OUTGOING_ALERTING**

outgoing call setup in alerting state

enum **esp_hf_roaming_status_t**

+CIND roaming status indicator values

Values:

enumerator **ESP_HF_ROAMING_STATUS_INACTIVE**

roaming is not active

enumerator **ESP_HF_ROAMING_STATUS_ACTIVE**

a roaming is active

enum **esp_hf_call_held_status_t**

+CIND call held indicator values

Values:

enumerator **ESP_HF_CALL_HELD_STATUS_NONE**

no calls held

enumerator **ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE**

both active and held call

enumerator **ESP_HF_CALL_HELD_STATUS_HELD**

call on hold, no active call

enum **esp_hf_current_call_status_t**

+CLCC status of the call

Values:

enumerator **ESP_HF_CURRENT_CALL_STATUS_ACTIVE**

active

enumerator **ESP_HF_CURRENT_CALL_STATUS_HELD**

held

enumerator **ESP_HF_CURRENT_CALL_STATUS_DIALING**

dialing (outgoing calls only)

enumerator **ESP_HF_CURRENT_CALL_STATUS_ALERTING**

alerting (outgoing calls only)

enumerator **ESP_HF_CURRENT_CALL_STATUS_INCOMING**

incoming (incoming calls only)

enumerator **ESP_HF_CURRENT_CALL_STATUS_WAITING**

waiting (incoming calls only)

enumerator **ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD**

call held by response and hold

enum **esp_hf_current_call_direction_t**

+CLCC direction of the call

Values:

enumerator **ESP_HF_CURRENT_CALL_DIRECTION_OUTGOING**

outgoing

enumerator **ESP_HF_CURRENT_CALL_DIRECTION_INCOMING**

incoming

enum **esp_hf_current_call_mpty_type_t**

+CLCC multi-party call flag

Values:

enumerator **ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE**

not a member of a multi-party call

enumerator **ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI**

member of a multi-party call

enum **esp_hf_current_call_mode_t**

+CLCC call mode

Values:

enumerator **ESP_HF_CURRENT_CALL_MODE_VOICE**

enumerator **ESP_HF_CURRENT_CALL_MODE_DATA**

enumerator **ESP_HF_CURRENT_CALL_MODE_FAX**

enum **esp_hf_call_addr_type_t**

+CLCC address type

Values:

enumerator **ESP_HF_CALL_ADDR_TYPE_UNKNOWN**

unkown address type

enumerator **ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL**

international address

enum **esp_hf_subscriber_service_type_t**

+CNUM service type of the phone number

Values:

enumerator **ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN**

unknown

enumerator **ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE**

voice service

enumerator **ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX**

fax service

enum **esp_hf_btrh_status_t**

+BTRH response and hold result code

Values:

enumerator **ESP_HF_BTRH_STATUS_HELD**
incoming call is put on held in AG

enumerator **ESP_HF_BTRH_STATUS_ACCEPTED**
held incoming call is accepted in AG

enumerator **ESP_HF_BTRH_STATUS_REJECTED**
held incoming call is rejected in AG

enum **esp_hf_btrh_cmd_t**
AT+BTRH response and hold action code.
Values:

enumerator **ESP_HF_BTRH_CMD_HOLD**
put the incoming call on hold

enumerator **ESP_HF_BTRH_CMD_ACCEPT**
accept a held incoming call

enumerator **ESP_HF_BTRH_CMD_REJECT**
reject a held incoming call

enum **esp_hf_nrec_t**
Values:

enumerator **ESP_HF_NREC_STOP**

enumerator **ESP_HF_NREC_START**

enum **esp_hf_call_waiting_status_t**
+CCWA resposne status
Values:

enumerator **ESP_HF_CALL_WAITING_INACTIVE**

enumerator **ESP_HF_CALL_WAITING_ACTIVE**

enum **esp_hf_wbs_config_t**
Values:

enumerator **ESP_HF_WBS_NONE**

enumerator **ESP_HF_WBS_NO**

enumerator **ESP_HF_WBS_YES**

enum **esp_hf_connection_state_t**

Bluetooth HFP RFCOMM connection and service level connection status.

Values:

enumerator **ESP_HF_CONNECTION_STATE_DISCONNECTED**

RFCOMM data link channel released

enumerator **ESP_HF_CONNECTION_STATE_CONNECTING**

connecting remote device on the RFCOMM data link

enumerator **ESP_HF_CONNECTION_STATE_CONNECTED**

RFCOMM connection established

enumerator **ESP_HF_CONNECTION_STATE_SLC_CONNECTED**

service level connection established

enumerator **ESP_HF_CONNECTION_STATE_DISCONNECTING**

disconnecting with remote device on the RFCOMM data link

enum **esp_hf_chld_type_t**

AT+CHLD command values.

Values:

enumerator **ESP_HF_CHLD_TYPE_REL**

<0>, Terminate all held or set UDUB(“busy”) to a waiting call

enumerator **ESP_HF_CHLD_TYPE_REL_ACC**

<1>, Terminate all active calls and accepts a waiting/held call

enumerator **ESP_HF_CHLD_TYPE_HOLD_ACC**

<2>, Hold all active calls and accepts a waiting/held call

enumerator **ESP_HF_CHLD_TYPE_MERGE**

<3>, Add all held calls to a conference

enumerator **ESP_HF_CHLD_TYPE_MERGE_DETACH**

<4>, connect the two calls and disconnects the subscriber from both calls

enumerator **ESP_HF_CHLD_TYPE_REL_X**

<1x>, releases specified calls only

enumerator **ESP_HF_CHLD_TYPE_PRIV_X**

<2x>, request private consultation mode with specified call

enum **esp_hf_at_response_code_t**

Values:

enumerator **ESP_HF_AT_RESPONSE_CODE_OK**
acknowledges execution of a command line

enumerator **ESP_HF_AT_RESPONSE_CODE_ERR**
command not accepted

enumerator **ESP_HF_AT_RESPONSE_CODE_NO_CARRIER**
connection terminated

enumerator **ESP_HF_AT_RESPONSE_CODE_BUSY**
busy signal detected

enumerator **ESP_HF_AT_RESPONSE_CODE_NO_ANSWER**
connection completion timeout

enumerator **ESP_HF_AT_RESPONSE_CODE_DELAYED**
delayed

enumerator **ESP_HF_AT_RESPONSE_CODE_BLACKLISTED**
blacklisted

enumerator **ESP_HF_AT_RESPONSE_CODE_CME**
CME error

enum **esp_hf_at_response_t**
Values:

enumerator **ESP_HF_AT_RESPONSE_ERROR**

enumerator **ESP_HF_AT_RESPONSE_OK**

enum **esp_hf_cme_err_t**
Extended Audio Gateway Error Result Code Response.
Values:

enumerator **ESP_HF_CME_AG_FAILURE**
ag failure

enumerator **ESP_HF_CME_NO_CONNECTION_TO_PHONE**
no connection to phone

enumerator **ESP_HF_CME_OPERATION_NOT_ALLOWED**
operation not allowed

enumerator **ESP_HF_CME_OPERATION_NOT_SUPPORTED**
operation not supported

enumerator **ESP_HF_CME_PH_SIM_PIN_REQUIRED**
PH-SIM PIN Required

enumerator **ESP_HF_CME_SIM_NOT_INSERTED**
SIM not inserted

enumerator **ESP_HF_CME_SIM_PIN_REQUIRED**
SIM PIN required

enumerator **ESP_HF_CME_SIM_PUK_REQUIRED**
SIM PUK required

enumerator **ESP_HF_CME_SIM_FAILURE**
SIM failure

enumerator **ESP_HF_CME_SIM_BUSY**
SIM busy

enumerator **ESP_HF_CME_INCORRECT_PASSWORD**
incorrect password

enumerator **ESP_HF_CME_SIM_PIN2_REQUIRED**
SIM PIN2 required

enumerator **ESP_HF_CME_SIM_PUK2_REQUIRED**
SIM PUK2 required

enumerator **ESP_HF_CME_MEMORY_FULL**
memory full

enumerator **ESP_HF_CME_INVALID_INDEX**
invalid index

enumerator **ESP_HF_CME_MEMORY_FAILURE**
memory failure

enumerator **ESP_HF_CME_TEXT_STRING_TOO_LONG**
test string too long

enumerator **ESP_HF_CME_INVALID_CHARACTERS_IN_TEXT_STRING**
invalid characters in text string

enumerator **ESP_HF_CME_DIAL_STRING_TOO_LONG**
dial string too long

enumerator **ESP_HF_CME_INVALID_CHARACTERS_IN_DIAL_STRING**
invalid characters in dial string

enumerator **ESP_HF_CME_NO_NETWORK_SERVICE**

no network service

enumerator **ESP_HF_CME_NETWORK_TIMEOUT**

network timeout

enumerator **ESP_HF_CME_NETWORK_NOT_ALLOWED**

network not allowed —emergency calls only

HFP CLIENT API

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_hf_client_api.h

Functions

esp_err_t **esp_hf_client_register_callback**(*esp_hf_client_cb_t* callback)

Register application callback function to HFP client module. This function should be called only after esp_bluedroid_enable() completes successfully.

Parameters **callback** –**[in]** HFP client event callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_hf_client_init**(void)

Initialize the bluetooth HFP client module. This function should be called after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: if the initialization request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_deinit**(void)

De-initialize for HFP client module. This function should be called only after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_connect**(*esp_bd_addr_t* remote_bda)

Establish a Service Level Connection to remote bluetooth HFP audio gateway(AG) device. This function must be called after esp_hf_client_init() and before esp_hf_client_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: connect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote HFP audio gateway. This function must be called after esp_hf_client_init() and before esp_hf_client_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: disconnect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_connect_audio** (*esp_bd_addr_t* remote_bda)

Create audio connection with remote HFP AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: connect audio request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_disconnect_audio** (*esp_bd_addr_t* remote_bda)

Release the established audio connection with remote HFP AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **remote_bda** –**[in]** remote bluetooth device address

Returns

- ESP_OK: disconnect audio request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_start_voice_recognition** (void)

Enable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: starting voice recognition is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_stop_voice_recognition** (void)

Disable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: stoping voice recognition is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_volume_update** (*esp_hf_volume_control_target_t* type, int volume)

Volume synchronization with AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters

- **type** –**[in]** volume control target, speaker or microphone
- **volume** –**[in]** gain of the speaker of microphone, ranges 0 to 15

Returns

- ESP_OK: volume update is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_dial** (const char *number)

Place a call with a specified number, if number is NULL, last called number is called. As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **number** –**[in]** number string of the call. If NULL, the last number is called(aka re-dial)

Returns

- ESP_OK: a call placing is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_dial_memory**(int location)

Place a call with number specified by location(speed dial). As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **location** –**[in]** location of the number in the memory

Returns

- ESP_OK: a memory call placing is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_chld_cmd**(*esp_hf_chld_type_t* chld, int idx)

Send call hold and multiparty commands, or enhanced call control commands(Use AT+CHLD). As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters

- **chld** –**[in]** AT+CHLD call hold and multiparty handling AT command.
- **idx** –**[in]** used in Enhanced Call Control Mechanisms, used if chld is ESP_HF_CHLD_TYPE_REL_X or ESP_HF_CHLD_TYPE_PRIV_X

Returns

- ESP_OK: command AT+CHLD is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_btrh_cmd**(*esp_hf_btrh_cmd_t* btrh)

Send response and hold action command(Send AT+BTRH command) As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **btrh** –**[in]** response and hold action to send

Returns

- ESP_OK: command AT+BTRH is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_answer_call**(void)

Answer an incoming call(send ATA command). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: a call answering is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_reject_call**(void)

Reject an incoming call or terminate an ongoing call(send AT+CHUP command). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: the call rejecting is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_query_current_calls**(void)

Query list of current calls in AG(send AT+CLCC command). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: query of current calls is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_query_current_operator_name**(void)

Query the name of currently selected network operator in AG(use AT+COPS commands). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: query of current operator name is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_retrieve_subscriber_info**(void)

Get subscriber information number from AG(send AT+CNUM command) As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: the retrieving of subscriber information is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_dtmf**(char code)

Transmit DTMF codes during an ongoing call(use AT+VTS commands) As a precondition to use this API, Service Level Connection shall exist with AG.

Parameters **code** –**[in]** dtmf code, single ascii character in the set 0-9, #, *, A-D

Returns

- ESP_OK: the DTMF codes are sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_xapl**(char *information, uint32_t features)

Send command to enable Vendor specific feature to indicate battery level and docker status This is Apple-specific commands, but used by most device, including Android and Windows.

Parameters

- **information** –**[in]** XAPL vendorID-productID-version, such as "0505-1995-0610" vendorID: A string representation of the hex value of the vendor ID from the manufacturer, without the 0x prefix. productID: A string representation of the hex value of the product ID from the manufacturer, without the 0x prefix. version: The revision of the software
- **features** –**[in]** A base-10 representation of a bit field. such as ESP_HF_CLIENT_XAPL_FEAT_BATTERY_REPORT

Returns

- ESP_OK: Feature enable request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_iphoneaccev**(uint32_t bat_level, bool docked)

Send Battery level and docker status Enable this feature using XAPL command first This is Apple-specific commands, but used by most device, including Android and Windows.

Parameters

- **bat_level** –**[in]** Battery Level: value between 0 and 9
- **docked** –**[in]** Dock State: false = undocked, true = docked

Returns

- ESP_OK: battery level is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_request_last_voice_tag_number**(void)

Request a phone number from AG corresponding to last voice tag recorded (send AT+BINP command). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: the phone number request corresponding to last voice tag recorded is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_nrec**(void)

Disable echo cancellation and noise reduction in the AG (use AT+NREC=0 command). As a precondition to use this API, Service Level Connection shall exist with AG.

Returns

- ESP_OK: NREC=0 request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_register_data_callback**(*esp_hf_client_incoming_data_cb_t* recv, *esp_hf_client_outgoing_data_cb_t* send)

Register HFP client data output function; the callback is only used in the case that Voice Over HCI is enabled.

Parameters

- **recv** –**[in]** HFP client incoming data callback function
- **send** –**[in]** HFP client outgoing data callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_hf_client_pkt_stat_nums_get**(uint16_t sync_conn_handle)

Get the number of packets received and sent This function is only used in the case that Voice Over HCI is enabled and the audio state is connected. When the operation is completed, the callback function will be called with ESP_HF_CLIENT_PKT_STAT_NUMS_GET_EVT.

Parameters **sync_conn_handle** –**[in]** the (e)SCO connection handle

Returns

- ESP_OK: if the request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

void **esp_hf_client_outgoing_data_ready**(void)

Trigger the lower-layer to fetch and send audio data. This function is only only used in the case that Voice Over HCI is enabled. After this function is called, lower layer will invoke esp_hf_client_outgoing_data_cb_t to fetch data.

As a precondition to use this API, Service Level Connection shall exist with AG.

void **esp_hf_client_pcm_resample_init**(uint32_t src_sps, uint32_t bits, uint32_t channels)

Initialize the down sampling converter. This is a utility function that can only be used in the case that Voice Over HCI is enabled.

Parameters

- **src_sps** –**[in]** original samples per second(source audio data, i.e. 48000, 32000, 16000, 44100, 22050, 11025)
- **bits** –**[in]** number of bits per pcm sample (16)
- **channels** –**[in]** number of channels (i.e. mono(1), stereo(2)···)

void **esp_hf_client_pcm_resample_deinit**(void)

Deinitialize the down sampling converter.

int32_t **esp_hf_client_pcm_resample**(void *src, uint32_t in_bytes, void *dst)

Down sampling utility to convert high sampling rate into 8K/16bits 1-channel mode PCM samples. This can only be used in the case that Voice Over HCI is enabled.

Parameters

- **src** –**[in]** pointer to the buffer where the original sampling PCM are stored
- **in_bytes** –**[in]** length of the input PCM sample buffer in byte
- **dst** –**[in]** pointer to the buffer which is to be used to store the converted PCM samples

Returns number of samples converted

Unions

union **esp_hf_client_cb_param_t**

#include <esp_hf_client_api.h> HFP client callback parameters.

Public Members

struct *esp_hf_client_cb_param_t*::*hf_client_conn_stat_param* **conn_stat**

HF callback param of ESP_HF_CLIENT_CONNECTION_STATE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_audio_stat_param* **audio_stat**

HF callback param of ESP_HF_CLIENT_AUDIO_STATE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_bvra_param* **bvra**

HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_service_availability_param* **service_availability**

HF callback param of ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_network_roaming_param* **roaming**

HF callback param of ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_signal_strength_ind_param* **signal_strength**

HF callback param of ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_battery_level_ind_param* **battery_level**

HF callback param of ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_current_operator_param* **cops**

HF callback param of ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_ind_param* **call**

HF callback param of ESP_HF_CLIENT_CIND_CALL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_setup_ind_param* **call_setup**

HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_held_ind_param* **call_held**

HF callback param of ESP_HF_CLIENT_CIND_CALL_HELD_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_btrh_param* **btrh**

HF callback param of ESP_HF_CLIENT_BRTH_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_clip_param* **clip**

HF callback param of ESP_HF_CLIENT_CLIP_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_ccwa_param* **ccwa**

HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_clcc_param* **clcc**

HF callback param of ESP_HF_CLIENT_CLCC_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_volume_control_param* **volume_control**

HF callback param of ESP_HF_CLIENT_VOLUME_CONTROL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_at_response_param* **at_response**

HF callback param of ESP_HF_CLIENT_AT_RESPONSE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_cnum_param* **cnum**

HF callback param of ESP_HF_CLIENT_CNUM_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_bsirparam* **bsir**

HF callback param of ESP_HF_CLIENT_BSIR_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_binp_param* **binp**

HF callback param of ESP_HF_CLIENT_BINP_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_pkt_status_nums* **pkt_nums**

HF callback param of ESP_HF_CLIENT_PKT_STAT_NUMS_GET_EVT

struct **hf_client_at_response_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_AT_RESPONSE_EVT.

Public Members

esp_hf_at_response_code_t **code**

AT response code

esp_hf_cme_err_t **cme**

Extended Audio Gateway Error Result Code

struct **hf_client_audio_stat_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_AUDIO_STATE_EVT.

Public Members

esp_hf_client_audio_state_t **state**

audio connection state

esp_bd_addr_t **remote_bda**

remote bluetooth device address

uint16_t **sync_conn_handle**

(e)SCO connection handle

struct **hf_client_battery_level_ind_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT.

Public Members

int **value**

battery charge value, ranges from 0 to 5

struct **hf_client_binp_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_BINP_EVT.

Public Members

const char ***number**

phone number corresponding to the last voice tag in the HF

struct **hf_client_bsirparam**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_BSIR_EVT.

Public Members

esp_hf_client_in_band_ring_state_t **state**

setting state of in-band ring tone

struct **hf_client_btrh_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_BTRH_EVT.

Public Members

esp_hf_btrh_status_t **status**

call hold and response status result code

struct **hf_client_bvra_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_BVRA_EVT.

Public Members

esp_hf_vr_state_t **value**

voice recognition state

struct **hf_client_call_held_ind_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_HELD_EVT.

Public Members

esp_hf_call_held_status_t **status**

bluetooth proprietary call hold status indicator

struct **hf_client_call_ind_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_EVT.

Public Members

esp_hf_call_status_t **status**

call status indicator

struct **hf_client_call_setup_ind_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_SETUP_EVT.

Public Members

esp_hf_call_setup_status_t **status**

call setup status indicator

struct **hf_client_ccwa_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CCWA_EVT.

Public Members

const char ***number**

phone number string of waiting call

struct **hf_client_clcc_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CLCC_EVT.

Public Members

int **idx**

numbering(starting with 1) of the call

esp_hf_current_call_direction_t **dir**

direction of the call

esp_hf_current_call_status_t **status**

status of the call

esp_hf_current_call_mpty_type_t **mpty**

multi-party flag

char ***number**

phone number(optional)

struct **hf_client_clip_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CLIP_EVT.

Public Members

const char ***number**

phone number string of call

struct **hf_client_cnum_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CNUM_EVT.

Public Members

const char ***number**

phone number string

esp_hf_subscriber_service_type_t **type**

service type that the phone number relates to

struct **hf_client_conn_stat_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CONNECTION_STATE_EVT.

Public Members

esp_hf_client_connection_state_t **state**

HF connection state

uint32_t **peer_feat**

AG supported features

uint32_t **chld_feat**

AG supported features on call hold and multiparty services

esp_bd_addr_t **remote_bda**

remote bluetooth device address

struct **hf_client_current_operator_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT.

Public Members

const char ***name**

name of the network operator

struct **hf_client_network_roaming_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT.

Public Members

esp_hf_roaming_status_t **status**

roaming status

struct **hf_client_pkt_status_nums**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_PKT_STAT_NUMS_GET_EVT.

Public Members

uint32_t **rx_total**

the total number of packets received

uint32_t **rx_correct**

the total number of packets data correctly received

uint32_t **rx_err**

the total number of packets data with possible invalid

uint32_t **rx_none**

the total number of packets data no received

uint32_t **rx_lost**

the total number of packets data partially lost

uint32_t **tx_total**

the total number of packets send

uint32_t **tx_discarded**

the total number of packets send lost

struct **hf_client_service_availability_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT.

Public Members

esp_hf_network_state_t **status**

service availability status

struct **hf_client_signal_strength_ind_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT.

Public Members

int **value**

signal strength value, ranges from 0 to 5

struct **hf_client_volume_control_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_VOLUME_CONTROL_EVT.

Public Members

esp_hf_volume_control_target_t **type**

volume control target, speaker or microphone

int **volume**

gain, ranges from 0 to 15

Macros

ESP_BT_HF_CLIENT_NUMBER_LEN

ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN

ESP_BT_HF_AT_SEND_XAPL_LEN

ESP_HF_CLIENT_PEER_FEAT_3WAY

ESP_HF_CLIENT_PEER_FEAT_ECNR

ESP_HF_CLIENT_PEER_FEAT_VREC

ESP_HF_CLIENT_PEER_FEAT_INBAND

ESP_HF_CLIENT_PEER_FEAT_VTAG

ESP_HF_CLIENT_PEER_FEAT_REJECT

ESP_HF_CLIENT_PEER_FEAT_ECS

ESP_HF_CLIENT_PEER_FEAT_ECC

ESP_HF_CLIENT_PEER_FEAT_EXTERR

ESP_HF_CLIENT_PEER_FEAT_CODEC

ESP_HF_CLIENT_PEER_FEAT_HF_IND

ESP_HF_CLIENT_PEER_FEAT_ESCO_S4

ESP_HF_CLIENT_CHLD_FEAT_REL

ESP_HF_CLIENT_CHLD_FEAT_REL_ACC

ESP_HF_CLIENT_CHLD_FEAT_REL_X

ESP_HF_CLIENT_CHLD_FEAT_HOLD_ACC

ESP_HF_CLIENT_CHLD_FEAT_PRIV_X

ESP_HF_CLIENT_CHLD_FEAT_MERGE

ESP_HF_CLIENT_CHLD_FEAT_MERGE_DETACH

ESP_HF_CLIENT_XAPL_FEAT_RESERVED

ESP_HF_CLIENT_XAPL_FEAT_BATTERY_REPORT

ESP_HF_CLIENT_XAPL_FEAT_DOCKED

ESP_HF_CLIENT_XAPL_FEAT_SIRI_STATUS_REPORT

ESP_HF_CLIENT_XAPL_NR_STATUS_REPORT

Type Definitions

typedef void (*__esp_hf_client_incoming_data_cb_t__)(const uint8_t *buf, uint32_t len)

HFP client incoming data callback function, the callback is useful in case of Voice Over HCI.

Param buf [in] : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.

Param len [in] : size(in bytes) in buf

typedef uint32_t (*__esp_hf_client_outgoing_data_cb_t__)(uint8_t *buf, uint32_t len)

HFP client outgoing data callback function, the callback is useful in case of Voice Over HCI. Once audio connection is set up and the application layer has prepared data to send, the lower layer will call this function to read data and then send. This callback is supposed to be implemented as non-blocking, and if data is not enough, return value 0 is supposed.

Param buf [in] : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.

Param len [in] : size(in bytes) in buf

Return length of data successfully read

typedef void (*`esp_hf_client_cb_t`)(*esp_hf_client_cb_event_t* event, *esp_hf_client_cb_param_t* *param)

HFP client callback function type.

Param event : Event type
Param param : Pointer to callback parameter

Enumerations

enum `esp_hf_client_connection_state_t`

Bluetooth HFP RFCOMM connection and service level connection status.

Values:

enumerator `ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED`

RFCOMM data link channel released

enumerator `ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING`

connecting remote device on the RFCOMM data link

enumerator `ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED`

RFCOMM connection established

enumerator `ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED`

service level connection established

enumerator `ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING`

disconnecting with remote device on the RFCOMM dat link

enum `esp_hf_client_audio_state_t`

Bluetooth HFP audio connection status.

Values:

enumerator `ESP_HF_CLIENT_AUDIO_STATE_DISCONNECTED`

audio connection released

enumerator `ESP_HF_CLIENT_AUDIO_STATE_CONNECTING`

audio connection has been initiated

enumerator `ESP_HF_CLIENT_AUDIO_STATE_CONNECTED`

audio connection is established

enumerator `ESP_HF_CLIENT_AUDIO_STATE_CONNECTED_MSBC`

mSBC audio connection is established

enum `esp_hf_client_in_band_ring_state_t`

in-band ring tone state

Values:

enumerator `ESP_HF_CLIENT_IN_BAND_RINGTONE_NOT_PROVIDED`

enumerator **ESP_HF_CLIENT_IN_BAND_RINGTONE_PROVIDED**

enum **esp_hf_client_cb_event_t**

HF CLIENT callback events.

Values:

enumerator **ESP_HF_CLIENT_CONNECTION_STATE_EVT**

connection state changed event

enumerator **ESP_HF_CLIENT_AUDIO_STATE_EVT**

audio connection state change event

enumerator **ESP_HF_CLIENT_BVRA_EVT**

voice recognition state change event

enumerator **ESP_HF_CLIENT_CIND_CALL_EVT**

call indication

enumerator **ESP_HF_CLIENT_CIND_CALL_SETUP_EVT**

call setup indication

enumerator **ESP_HF_CLIENT_CIND_CALL_HELD_EVT**

call held indication

enumerator **ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT**

network service availability indication

enumerator **ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT**

signal strength indication

enumerator **ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT**

roaming status indication

enumerator **ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT**

battery level indication

enumerator **ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT**

current operator information

enumerator **ESP_HF_CLIENT_BTRH_EVT**

call response and hold event

enumerator **ESP_HF_CLIENT_CLIP_EVT**

Calling Line Identification notification

enumerator **ESP_HF_CLIENT_CCWA_EVT**

call waiting notification

enumerator **ESP_HF_CLIENT_CLCC_EVT**

list of current calls notification

enumerator **ESP_HF_CLIENT_VOLUME_CONTROL_EVT**

audio volume control command from AG, provided by +VGM or +VGS message

enumerator **ESP_HF_CLIENT_AT_RESPONSE_EVT**

AT command response event

enumerator **ESP_HF_CLIENT_CNUM_EVT**

subscriber information response from AG

enumerator **ESP_HF_CLIENT_BSIR_EVT**

setting of in-band ring tone

enumerator **ESP_HF_CLIENT_BINP_EVT**

requested number of last voice tag from AG

enumerator **ESP_HF_CLIENT_RING_IND_EVT**

ring indication event

enumerator **ESP_HF_CLIENT_PKT_STAT_NUMS_GET_EVT**

requested number of packet different status

HFP AG API

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_hf_ag_api.h

Functions

esp_err_t **esp_hf_ag_register_callback**(*esp_hf_cb_t* callback)

Register application callback function to HFP AG module. This function should be called only after esp_bluedroid_enable() completes successfully.

Parameters **callback** –**[in]** HFP AG event callback function

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_hf_ag_init**(void)

Initialize the bluetooth HF AG module. This function should be called after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: if the initialization request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_deinit** (void)

De-initialize for HF AG module. This function should be called only after esp_bluedroid_enable() completes successfully.

Returns
- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_slc_connect** (*esp_bd_addr_t* remote_bda)

To establish a Service Level Connection to remote bluetooth HFP client device. This function must be called after esp_hf_ag_init() and before esp_hf_ag_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth HFP client device address
Returns
- ESP_OK: connect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_slc_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote HFP client. This function must be called after esp_hf_ag_init() and before esp_hf_ag_deinit().

Parameters **remote_bda** –**[in]** remote bluetooth device address
Returns
- ESP_OK: disconnect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_audio_connect** (*esp_bd_addr_t* remote_bda)

Create audio connection with remote HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters **remote_bda** –**[in]** remote bluetooth device address
Returns
- ESP_OK: audio connect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_audio_disconnect** (*esp_bd_addr_t* remote_bda)

Release the established audio connection with remote HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters **remote_bda** –**[in]** remote bluetooth device address
Returns
- ESP_OK: audio disconnect request is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_vra_control** (*esp_bd_addr_t* remote_bda, *esp_hf_vr_state_t* value)

Response of Volume Recognition Command(AT+VRA) from HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters
- **remote_bda** –**[in]** the device address of voice recognition initiator
- **value** –**[in]** 0 - voice recognition disabled, 1- voice recognition enabled

Returns
- ESP_OK: response of volume recognition is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_volume_control** (*esp_bd_addr_t* remote_bda, *esp_hf_volume_control_target_t* type, int volume)

Volume synchronization with HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_bda** –**[in]** remote bluetooth device address
- **type** –**[in]** volume control target, speaker or microphone
- **volume** –**[in]** gain of the speaker of microphone, ranges 0 to 15

Returns

- ESP_OK: volume synchronization control is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if arguments are invalid
- ESP_FAIL: others

esp_err_t **esp_hf_ag_unknown_at_send** (*esp_bd_addr_t* remote_addr, char *unat)

Handle Unknown AT command from HFP Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **unat** –**[in]** User AT command response to HF Client. It will response "ERROR" by default if unat is NULL.

Returns

- ESP_OK: response of unknown AT command is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_cmee_send** (*esp_bd_addr_t* remote_bda, *esp_hf_at_response_code_t* response_code, *esp_hf_cme_err_t* error_code)

Unsolicited send extend AT error code to HFP Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_bda** –**[in]** remote bluetooth device address
- **response_code** –**[in]** AT command response code
- **error_code** –**[in]** CME error code

Returns

- ESP_OK: extend error code is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_devices_status_indchange** (*esp_bd_addr_t* remote_addr, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, *esp_hf_network_state_t* ntk_state, int signal)

Unsolicited send device status notification to HFP Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **ntk_state** –**[in]** network service state
- **signal** –**[in]** signal strength from 0 to 5

Returns

- ESP_OK: device status notification is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if arguments are invalid
- ESP_FAIL: others

esp_err_t **esp_hf_ag_ciev_report**(*esp_bd_addr_t* remote_addr, *esp_hf_ciev_report_type_t* ind_type, int value)

Send indicator report "+CIEV: <ind> <value>" to HFP Client. "CIEV" means "indicator events reporting" , and all indicator types can be sent one type at a time. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **ind_type** –**[in]** indicator type
- **value** –**[in]** indicator value

Returns

- ESP_OK: indicator report is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_cind_response**(*esp_bd_addr_t* remote_addr, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, *esp_hf_network_state_t* ntk_state, int signal, *esp_hf_roaming_status_t* roam, int batt_lev, *esp_hf_call_held_status_t* call_held_status)

Response to device individual indicators to HFP Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **ntk_state** –**[in]** network service state
- **signal** –**[in]** signal strength from 0 to 5
- **roam** –**[in]** roam state
- **batt_lev** –**[in]** battery level from 0 to 5
- **call_held_status** –**[in]** call held status

Returns

- ESP_OK: response to device individual indicators is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if the arguments are invalid
- ESP_FAIL: others

esp_err_t **esp_hf_ag_cops_response**(*esp_bd_addr_t* remote_addr, char *name)

Reponse for AT+COPS command from HF Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **name** –**[in]** current operator name

Returns

- ESP_OK: reponse for AT+COPS command is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_clcc_response**(*esp_bd_addr_t* remote_addr, int index, *esp_hf_current_call_direction_t* dir, *esp_hf_current_call_status_t* current_call_state, *esp_hf_current_call_mode_t* mode, *esp_hf_current_call_mpty_type_t* mpty, char *number, *esp_hf_call_addr_type_t* type)

Response to AT+CLCC command from HFP Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **index** –**[in]** the index of current call, starting with 1, finishing response with 0 (send OK)

- **dir** –**[in]** call direction (incoming/outgoing)
- **current_call_state** –**[in]** current call state
- **mode** –**[in]** current call mode (voice/data/fax)
- **mpty** –**[in]** single or multi type
- **number** –**[in]** current call number
- **type** –**[in]** international type or unknow

Returns

- ESP_OK: response to AT+CLCC command is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_cnum_response**(*esp_bd_addr_t* remote_addr, char *number, int number_type, *esp_hf_subscriber_service_type_t* service_type)

Response for AT+CNUM command from HF Client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **number** –**[in]** registration number
- **number_type** –**[in]** value of number type from 128-143: national or international, may contain prefix and/or escape digits 144-159: international, includes country code prefix, add "+" if needed 160-175: national, but no prefix nor escape digits
- **service_type** –**[in]** service type (unknown/voice/fax)

Returns

- ESP_OK: response for AT+CNUM command is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_bsir**(*esp_bd_addr_t* remote_addr, *esp_hf_in_band_ring_state_t* state)

Inform HF Client that AG Provided in-band ring tone or not. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **state** –**[in]** in-band ring tone state

Returns

- ESP_OK: information of in-band ring tone is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if arguments are invalid
- ESP_FAIL: others

esp_err_t **esp_hf_ag_answer_call**(*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Answer Incoming Call from AG. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **num_active** –**[in]** the number of active call
- **num_held** –**[in]** the number of held call
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **number** –**[in]** number of the incoming call
- **call_addr_type** –**[in]** call address type

Returns

- ESP_OK: answer incoming call is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_reject_call**(*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Reject Incoming Call from AG. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **num_active** –**[in]** the number of active call
- **num_held** –**[in]** the number of held call
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **number** –**[in]** number of the incoming call
- **call_addr_type** –**[in]** call address type

Returns

- ESP_OK: reject incoming call is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_out_call**(*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Initiate a call from AG. As a precondition to use this API, Service Level Connection shall exist with HFP client. If the AG is driven by the HF to call esp_hf_ag_out_call, it needs to response an OK or ERROR to HF. But if the AG is actively calling esp_hf_ag_out_call, it does not need to take a response to HF.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **num_active** –**[in]** the number of active call
- **num_held** –**[in]** the number of held call
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **number** –**[in]** number of the outgoing call
- **call_addr_type** –**[in]** call address type

Returns

- ESP_OK: a call initiation is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_end_call**(*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

End an ongoing call. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Parameters

- **remote_addr** –**[in]** remote bluetooth device address
- **num_active** –**[in]** the number of active call
- **num_held** –**[in]** the number of held call
- **call_state** –**[in]** call state
- **call_setup_state** –**[in]** call setup state
- **number** –**[in]** number of the call
- **call_addr_type** –**[in]** call address type

Returns

- ESP_OK: end an ongoing call is sent to lower layer
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_ag_register_data_callback**(*esp_hf_incoming_data_cb_t* recv, *esp_hf_outgoing_data_cb_t* send)

Register AG data output function. The callback is only used in the case that Voice Over HCI is enabled.

Parameters
- **recv** –**[in]** HFP client incoming data callback function
- **send** –**[in]** HFP client outgoing data callback function

Returns
- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

esp_err_t **esp_hf_ag_pkt_stat_nums_get**(uint16_t sync_conn_handle)

Get the number of packets received and sent.

```
            This function is only used in the case that Voice Over HCI is␣
↪enabled and the audio state is connected.
            When the operation is completed, the callback function will be␣
↪called with ESP_HF_PKT_STAT_NUMS_GET_EVT.
```

Parameters **sync_conn_handle** –**[in]** the (e)SCO connection handle

Returns
- ESP_OK: if the request is sent successfully
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

void **esp_hf_ag_outgoing_data_ready**(void)

Trigger the lower-layer to fetch and send audio data.

```
            This function is only used in the case that Voice Over HCI is␣
↪enabled.
            As a precondition to use this API, Service Level Connection shall␣
↪exist with HFP client.
            After this function is called, lower layer will invoke esp_hf_
↪client_outgoing_data_cb_t to fetch data
```

Unions

union **esp_hf_cb_param_t**

#include <esp_hf_ag_api.h> HFP AG callback parameters.

Public Members

struct *esp_hf_cb_param_t*::*hf_conn_stat_param* **conn_stat**

AG callback param of ESP_HF_CONNECTION_STATE_EVT

struct *esp_hf_cb_param_t*::*hf_audio_stat_param* **audio_stat**

AG callback param of ESP_HF_AUDIO_STATE_EVT

struct *esp_hf_cb_param_t*::*hf_vra_rep_param* **vra_rep**

AG callback param of ESP_HF_BVRA_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_volume_control_param* **volume_control**

AG callback param of ESP_HF_VOLUME_CONTROL_EVT

struct *esp_hf_cb_param_t*::*hf_unat_rep_param* **unat_rep**

AG callback param of ESP_HF_UNAT_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_out_call_param* **out_call**

AG callback param of ESP_HF_DIAL_EVT

struct *esp_hf_cb_param_t*::*hf_ind_upd_param* **ind_upd**

AG callback param of ESP_HF_IND_UPDATE_EVT

struct *esp_hf_cb_param_t*::*hf_cind_rep_param* **cind_rep**

AG callback param of ESP_HF_CIND_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_cops_rep_param* **cops_rep**

AG callback param of ESP_HF_COPS_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_clcc_rep_param* **clcc_rep**

AG callback param of ESP_HF_CLCC_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_cnum_rep_param* **cnum_rep**

AG callback param of ESP_HF_CNUM_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_vts_rep_param* **vts_rep**

AG callback param of ESP_HF_VTS_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_nrec_param* **nrec**

AG callback param of ESP_HF_NREC_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_ata_rep_param* **ata_rep**

AG callback param of ESP_HF_ATA_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_chup_rep_param* **chup_rep**

AG callback param of ESP_HF_CHUP_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_wbs_rep_param* **wbs_rep**

AG callback param of ESP_HF_WBS_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*hf_bcs_rep_param* **bcs_rep**

AG callback param of ESP_HF_BCS_RESPONSE_EVT

struct *esp_hf_cb_param_t*::*ag_pkt_status_nums* **pkt_nums**

AG callback param of ESP_HF_PKT_STAT_NUMS_GET_EVT

struct **ag_pkt_status_nums**

#include <esp_hf_ag_api.h> ESP_HF_PKT_STAT_NUMS_GET_EVT.

Public Members

uint32_t **rx_total**
the total number of packets received

uint32_t **rx_correct**
the total number of packets data correctly received

uint32_t **rx_err**
the total number of packets data with possible invalid

uint32_t **rx_none**
the total number of packets data no received

uint32_t **rx_lost**
the total number of packets data partially lost

uint32_t **tx_total**
the total number of packets send

uint32_t **tx_discarded**
the total number of packets send lost

struct **hf_ata_rep_param**
#include <esp_hf_ag_api.h> ESP_HF_ATA_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**
remote bluetooth device address

struct **hf_audio_stat_param**
#include <esp_hf_ag_api.h> ESP_HF_AUDIO_STATE_EVT.

Public Members

esp_bd_addr_t **remote_addr**
Remote bluetooth device address

esp_hf_audio_state_t **state**
Audio connection state

uint16_t **sync_conn_handle**
(e)SCO connection handle

struct **hf_bcs_rep_param**
#include <esp_hf_ag_api.h> ESP_HF_BCS_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

esp_hf_wbs_config_t **mode**

codec mode CVSD or mSBC

struct **hf_chup_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_CHUP_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_cind_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_CIND_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_clcc_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_CLCC_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_cnum_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_CNUM_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_conn_stat_param**

#include <esp_hf_ag_api.h> ESP_HF_CONNECTION_STATE_EVT.

Public Members

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

esp_hf_connection_state_t **state**

Connection state

uint32_t **peer_feat**

HF supported features

uint32_t **chld_feat**

AG supported features on call hold and multiparty services

struct **hf_cops_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_COPS_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_ind_upd_param**

#include <esp_hf_ag_api.h> ESP_HF_IND_UPDATE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

struct **hf_nrec_param**

#include <esp_hf_ag_api.h> ESP_HF_NREC_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

esp_hf_nrec_t **state**

NREC enabled or disabled

struct **hf_out_call_param**

#include <esp_hf_ag_api.h> ESP_HF_DIAL_EVT.

Public Members

esp_bd_addr_t **remote_addr**

remote bluetooth device address

esp_hf_dial_type_t **type**

dial type

char ***num_or_loc**

location in phone memory

struct **hf_unat_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_UNAT_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

char ***unat**

Unknown AT command string

struct **hf_volume_control_param**

#include <esp_hf_ag_api.h> ESP_HF_VOLUME_CONTROL_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

esp_hf_volume_type_t **type**

Volume control target, speaker or microphone

int **volume**

Gain, ranges from 0 to 15

struct **hf_vra_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_BVRA_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

esp_hf_vr_state_t **value**

Voice recognition state

struct **hf_vts_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_VTS_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

char ***code**

MTF code from HF Client

struct **hf_wbs_rep_param**

#include <esp_hf_ag_api.h> ESP_HF_WBS_RESPONSE_EVT.

Public Members

esp_bd_addr_t **remote_addr**

Remote bluetooth device address

esp_hf_wbs_config_t **codec**

codec mode CVSD or mSBC

Macros

ESP_HF_PEER_FEAT_3WAY

ESP_HF_PEER_FEAT_ECNR

ESP_HF_PEER_FEAT_VREC

ESP_HF_PEER_FEAT_INBAND

ESP_HF_PEER_FEAT_VTAG

ESP_HF_PEER_FEAT_REJECT

ESP_HF_PEER_FEAT_ECS

ESP_HF_PEER_FEAT_ECC

ESP_HF_PEER_FEAT_EXTERR

ESP_HF_PEER_FEAT_CODEC

ESP_HF_PEER_FEAT_HF_IND

ESP_HF_PEER_FEAT_ESCO_S4

ESP_HF_CHLD_FEAT_REL

ESP_HF_CHLD_FEAT_REL_ACC

ESP_HF_CHLD_FEAT_REL_X

ESP_HF_CHLD_FEAT_HOLD_ACC

ESP_HF_CHLD_FEAT_PRIV_X

ESP_HF_CHLD_FEAT_MERGE

ESP_HF_CHLD_FEAT_MERGE_DETACH

Type Definitions

typedef void (***esp_hf_incoming_data_cb_t**)(const uint8_t *buf, uint32_t len)

AG incoming data callback function, the callback is useful in case of Voice Over HCI.

Param buf **[in]** : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
Param len **[in]** : size(in bytes) in buf

typedef uint32_t (***esp_hf_outgoing_data_cb_t**)(uint8_t *buf, uint32_t len)

AG outgoing data callback function, the callback is useful in case of Voice Over HCI. Once audio connection is set up and the application layer has prepared data to send, the lower layer will call this function to read data and then send. This callback is supposed to be implemented as non-blocking, and if data is not enough, return value 0 is supposed.

Param buf **[in]** : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
Param len **[in]** : size(in bytes) in buf
Return length of data successfully read

typedef void (***esp_hf_cb_t**)(*esp_hf_cb_event_t* event, *esp_hf_cb_param_t* *param)

HF AG callback function type.

Param event : Event type
Param param : Pointer to callback parameter

Enumerations

enum **esp_hf_cb_event_t**

HF callback events.

Values:

enumerator **ESP_HF_CONNECTION_STATE_EVT**

Connection state changed event

enumerator **ESP_HF_AUDIO_STATE_EVT**

Audio connection state change event

enumerator **ESP_HF_BVRA_RESPONSE_EVT**

Voice recognition state change event

enumerator **ESP_HF_VOLUME_CONTROL_EVT**

Audio volume control command from HF Client, provided by +VGM or +VGS message

enumerator **ESP_HF_UNAT_RESPONSE_EVT**

Unknown AT cmd Response

enumerator **ESP_HF_IND_UPDATE_EVT**

Indicator Update Event

enumerator **ESP_HF_CIND_RESPONSE_EVT**

Call And Device Indicator Response

enumerator **ESP_HF_COPS_RESPONSE_EVT**

Current operator information

enumerator **ESP_HF_CLCC_RESPONSE_EVT**

List of current calls notification

enumerator **ESP_HF_CNUM_RESPONSE_EVT**

Subscriber information response from HF Client

enumerator **ESP_HF_VTS_RESPONSE_EVT**

Enable or not DTMF

enumerator **ESP_HF_NREC_RESPONSE_EVT**

Enable or not NREC

enumerator **ESP_HF_ATA_RESPONSE_EVT**

Answer an Incoming Call

enumerator **ESP_HF_CHUP_RESPONSE_EVT**

Reject an Incoming Call

enumerator **ESP_HF_DIAL_EVT**

Origin an outgoing call with specific number or the dial the last number

enumerator **ESP_HF_WBS_RESPONSE_EVT**

Codec Status

enumerator **ESP_HF_BCS_RESPONSE_EVT**

Final Codec Choice

enumerator **ESP_HF_PKT_STAT_NUMS_GET_EVT**

Request number of packet different status

enum **esp_hf_dial_type_t**

Dial type of ESP_HF_DIAL_EVT.

Values:

enumerator **ESP_HF_DIAL_NUM**

Dial with a phone number

enumerator **ESP_HF_DIAL_VOIP**

Dial with VoIP

enumerator **ESP_HF_DIAL_MEM**

Dial with a memory position

Bluetooth HID Device API

Overview A Bluetooth HID device is a device providing the service of human or other data input and output to and from a Bluetooth HID Host. Users can use the Bluetooth HID Device APIs to make devices like keyboards, mice, joysticks and so on.

Application Example Check bluetooth/bluedroid/classic_bt folder in ESP-IDF examples, which contains the following application:

- This is an example of Bluetooth HID mouse device. The device running this example can be discovered and connected by a Bluetooth HID Host device such as a PC, and the pointer will move left and right after HID connection is established - bluetooth/bluedroid/classic_bt/bt_hid_mouse_device

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_hidd_api.h

Functions

esp_err_t **esp_bt_hid_device_register_callback**(*esp_hd_cb_t* callback)

This function is called to init callbacks with HID device module.

Parameters **callback** –**[in]** pointer to the init callback function.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_init**(void)

Initializes HIDD interface. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_register_callback. When the operation is complete, the callback function will be called with ESP_HIDD_INIT_EVT.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_deinit**(void)

De-initializes HIDD interface. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_DEINIT_EVT.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_register_app**(*esp_hidd_app_param_t* *app_param, *esp_hidd_qos_param_t* *in_qos, *esp_hidd_qos_param_t* *out_qos)

Registers HIDD parameters with SDP and sets l2cap Quality of Service. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_REGISTER_APP_EVT.

Parameters
- **app_param** –**[in]** HIDD parameters
- **in_qos** –**[in]** incoming QoS parameters
- **out_qos** –**[in]** outgoing QoS parameters

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_unregister_app**(void)

Removes HIDD parameters from SDP and resets l2cap Quality of Service. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_UNREGISTER_APP_EVT.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_connect**(*esp_bd_addr_t* bd_addr)

Connects to the peer HID Host with virtual cable. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_OPEN_EVT.

Parameters **bd_addr** –**[in]** Remote host bluetooth device address.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_disconnect**(void)

Disconnects from the currently connected HID Host. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_CLOSE_EVT.

Note: The disconnect operation will not remove the virtually cabled device. If the connect request from the different HID Host, it will reject the request.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_send_report**(*esp_hidd_report_type_t* type, uint8_t id, uint16_t len, uint8_t *data)

Sends HID report to the currently connected HID Host. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_SEND_REPORT_EVT.

Parameters
- **type** –**[in]** type of report
- **id** –**[in]** report id as defined by descriptor
- **len** –**[in]** length of report
- **data** –**[in]** report data

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_report_error**(*esp_hidd_handshake_error_t* error)

Sends HID Handshake with error info for invalid set_report to the currently connected HID Host. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_REPORT_ERR_EVT.

Parameters **error** –**[in]** type of error

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_device_virtual_cable_unplug**(void)

Remove the virtually cabled device. This function should be called after esp_bluedroid_init() and esp_bluedroid_enable() success, and should be called after esp_bt_hid_device_init(). When the operation is complete, the callback function will be called with ESP_HIDD_VC_UNPLUG_EVT.

Note: If the connection exists, then HID Device will send a `VIRTUAL_CABLE_UNPLUG` control command to the peer HID Host, and the connection will be destroyed. If the connection does not exist, then HID Device will only unplug on it' s single side. Once the unplug operation is success, the related pairing and bonding information will be removed, then the HID Device can accept connection request from the different HID Host,

---

Returns - ESP_OK: success
- other: failed

Unions

union **esp_hidd_cb_param_t**

#include <esp_hidd_api.h> HID device callback parameters union.

Public Members

struct *esp_hidd_cb_param_t*::*hidd_init_evt_param* **init**

HIDD callback param of ESP_HIDD_INIT_EVT

struct *esp_hidd_cb_param_t*::*hidd_deinit_evt_param* **deinit**

HIDD callback param of ESP_HIDD_DEINIT_EVT

struct *esp_hidd_cb_param_t*::*hidd_register_app_evt_param* **register_app**

HIDD callback param of ESP_HIDD_REGISTER_APP_EVT

struct *esp_hidd_cb_param_t*::*hidd_unregister_app_evt_param* **unregister_app**

HIDD callback param of ESP_HIDD_UNREGISTER_APP_EVT

struct *esp_hidd_cb_param_t*::*hidd_open_evt_param* **open**

HIDD callback param of ESP_HIDD_OPEN_EVT

struct *esp_hidd_cb_param_t*::*hidd_close_evt_param* **close**

HIDD callback param of ESP_HIDD_CLOSE_EVT

struct *esp_hidd_cb_param_t*::*hidd_send_report_evt_param* **send_report**

HIDD callback param of ESP_HIDD_SEND_REPORT_EVT

struct *esp_hidd_cb_param_t*::*hidd_report_err_evt_param* **report_err**

HIDD callback param of ESP_HIDD_REPORT_ERR_EVT

struct *esp_hidd_cb_param_t*::*hidd_get_report_evt_param* **get_report**

HIDD callback param of ESP_HIDD_GET_REPORT_EVT

struct *esp_hidd_cb_param_t*::*hidd_set_report_evt_param* **set_report**

HIDD callback param of ESP_HIDD_SET_REPORT_EVT

struct *esp_hidd_cb_param_t*::*hidd_set_protocol_evt_param* **set_protocol**

HIDD callback param of ESP_HIDD_SET_PROTOCOL_EVT

struct *esp_hidd_cb_param_t*::*hidd_intr_data_evt_param* **intr_data**

HIDD callback param of ESP_HIDD_INTR_DATA_EVT

struct *esp_hidd_cb_param_t*::*hidd_vc_unplug_param* **vc_unplug**

HIDD callback param of ESP_HIDD_VC_UNPLUG_EVT

struct **hidd_close_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_CLOSE_EVT.

Public Members

esp_hidd_status_t **status**

operation status

esp_hidd_connection_state_t **conn_status**

connection status

struct **hidd_deinit_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_DEINIT_EVT.

Public Members

esp_hidd_status_t **status**
operation status

struct **hidd_get_report_evt_param**
#include <esp_hidd_api.h> ESP_HIDD_GET_REPORT_EVT.

Public Members

esp_hidd_report_type_t **report_type**
report type

uint8_t **report_id**
report id

uint16_t **buffer_size**
buffer size

struct **hidd_init_evt_param**
#include <esp_hidd_api.h> ESP_HIDD_INIT_EVT.

Public Members

esp_hidd_status_t **status**
operation status

struct **hidd_intr_data_evt_param**
#include <esp_hidd_api.h> ESP_HIDD_INTR_DATA_EVT.

Public Members

uint8_t **report_id**
interrupt channel report id

uint16_t **len**
interrupt channel report data length

uint8_t ***data**
interrupt channel report data pointer

struct **hidd_open_evt_param**
#include <esp_hidd_api.h> ESP_HIDD_OPEN_EVT.

Public Members

esp_hidd_status_t **status**
operation status

esp_hidd_connection_state_t **conn_status**

connection status

esp_bd_addr_t **bd_addr**

host address

struct **hidd_register_app_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_REGISTER_APP_EVT.

Public Members

esp_hidd_status_t **status**

operation status

bool **in_use**

indicate whether use virtual cable plug host address

esp_bd_addr_t **bd_addr**

host address

struct **hidd_report_err_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_REPORT_ERR_EVT.

Public Members

esp_hidd_status_t **status**

operation status

uint8_t **reason**

lower layer failed reason(ref hiddefs.h)

struct **hidd_send_report_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_SEND_REPORT_EVT.

Public Members

esp_hidd_status_t **status**

operation status

uint8_t **reason**

lower layer failed reason(ref hiddefs.h)

esp_hidd_report_type_t **report_type**

report type

uint8_t **report_id**

report id

struct **hidd_set_protocol_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_SET_PROTOCOL_EVT.

Public Members

esp_hidd_protocol_mode_t **protocol_mode**

protocol mode

struct **hidd_set_report_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_SET_REPORT_EVT.

Public Members

esp_hidd_report_type_t **report_type**

report type

uint8_t **report_id**

report id

uint16_t **len**

set_report data length

uint8_t ***data**

set_report data pointer

struct **hidd_unregister_app_evt_param**

#include <esp_hidd_api.h> ESP_HIDD_UNREGISTER_APP_EVT.

Public Members

esp_hidd_status_t **status**

operation status

struct **hidd_vc_unplug_param**

#include <esp_hidd_api.h> ESP_HIDD_VC_UNPLUG_EVT.

Public Members

esp_hidd_status_t **status**

operation status

esp_hidd_connection_state_t **conn_status**

connection status

Structures

struct **esp_hidd_app_param_t**

HID device characteristics for SDP server.

Public Members

const char ***name**

service name

const char ***description**

service description

const char ***provider**

provider name

uint8_t **subclass**

HID device subclass

uint8_t ***desc_list**

HID descriptor list

int **desc_list_len**

size in bytes of HID descriptor list

struct **esp_hidd_qos_param_t**

HIDD Quality of Service parameters negotiated over L2CAP.

Public Members

uint8_t **service_type**

the level of service, 0 indicates no traffic

uint32_t **token_rate**

token rate in bytes per second, 0 indicates “don’t care”

uint32_t **token_bucket_size**

limit on the burstness of the application data

uint32_t **peak_bandwidth**

bytes per second, value 0 indicates “don’t care”

uint32_t **access_latency**

maximum acceptable delay in microseconds

uint32_t **delay_variation**

the difference in microseconds between the max and min delay

Macros

ESP_HID_CLASS_UNKNOWN

subclass of hid device

unknown HID device subclass

ESP_HID_CLASS_JOS

joystick

ESP_HID_CLASS_GPD

game pad

ESP_HID_CLASS_RMC

remote control

ESP_HID_CLASS_SED

sensing device

ESP_HID_CLASS_DGT

digitizer tablet

ESP_HID_CLASS_CDR

card reader

ESP_HID_CLASS_KBD

keyboard

ESP_HID_CLASS_MIC

pointing device

ESP_HID_CLASS_COM

combo keyboard/pointing

Type Definitions

typedef void (***esp_hd_cb_t**)(*esp_hidd_cb_event_t* event, *esp_hidd_cb_param_t* *param)

HID device callback function type.

Param event Event type
Param param Point to callback parameter, currently is union type

Enumerations

enum **esp_hidd_handshake_error_t**

HIDD handshake result code.

Values:

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_SUCCESS**

successful

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_NOT_READY**
not ready, device is too busy to accept data

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_ERR_INVALID_REP_ID**
invalid report ID

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_ERR_UNSUPPORTED_REQ**
device does not support the request

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_ERR_INVALID_PARAM**
parameter value is out of range or inappropriate

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_ERR_UNKNOWN**
device could not identify the error condition

enumerator **ESP_HID_PAR_HANDSHAKE_RSP_ERR_FATAL**
restart is essential to resume functionality

enum **esp_hidd_report_type_t**
HIDD report types.
Values:

enumerator **ESP_HIDD_REPORT_TYPE_OTHER**
unknown report type

enumerator **ESP_HIDD_REPORT_TYPE_INPUT**
input report

enumerator **ESP_HIDD_REPORT_TYPE_OUTPUT**
output report

enumerator **ESP_HIDD_REPORT_TYPE_FEATURE**
feature report

enumerator **ESP_HIDD_REPORT_TYPE_INTRDATA**
special value for reports to be sent on interrupt channel, INPUT is assumed

enum **esp_hidd_connection_state_t**
HIDD connection state.
Values:

enumerator **ESP_HIDD_CONN_STATE_CONNECTED**
HID connection established

enumerator **ESP_HIDD_CONN_STATE_CONNECTING**
connection to remote Bluetooth device

enumerator **ESP_HIDD_CONN_STATE_DISCONNECTED**
connection released

enumerator **ESP_HIDD_CONN_STATE_DISCONNECTING**
disconnecting to remote Bluetooth device

enumerator **ESP_HIDD_CONN_STATE_UNKNOWN**
unknown connection state

enum **esp_hidd_protocol_mode_t**
HID device protocol modes.

Values:

enumerator **ESP_HIDD_REPORT_MODE**
Report Protocol Mode

enumerator **ESP_HIDD_BOOT_MODE**
Boot Protocol Mode

enumerator **ESP_HIDD_UNSUPPORTED_MODE**
unsupported

enum **esp_hidd_boot_report_id_t**
HID Boot Protocol report IDs.

Values:

enumerator **ESP_HIDD_BOOT_REPORT_ID_KEYBOARD**
report ID of Boot Protocol keyboard report

enumerator **ESP_HIDD_BOOT_REPORT_ID_MOUSE**
report ID of Boot Protocol mouse report

enum **[anonymous]**
HID Boot Protocol report size including report ID.

Values:

enumerator **ESP_HIDD_BOOT_REPORT_SIZE_KEYBOARD**
report size of Boot Protocol keyboard report

enumerator **ESP_HIDD_BOOT_REPORT_SIZE_MOUSE**
report size of Boot Protocol mouse report

enum **esp_hidd_cb_event_t**
HID device callback function events.

Values:

enumerator **ESP_HIDD_INIT_EVT**
When HID device is initialized, the event comes

enumerator **ESP_HIDD_DEINIT_EVT**
When HID device is deinitialized, the event comes

enumerator **ESP_HIDD_REGISTER_APP_EVT**

When HID device application registered, the event comes

enumerator **ESP_HIDD_UNREGISTER_APP_EVT**

When HID device application unregistered, the event comes

enumerator **ESP_HIDD_OPEN_EVT**

When HID device connection to host opened, the event comes

enumerator **ESP_HIDD_CLOSE_EVT**

When HID device connection to host closed, the event comes

enumerator **ESP_HIDD_SEND_REPORT_EVT**

When HID device send report to lower layer, the event comes

enumerator **ESP_HIDD_REPORT_ERR_EVT**

When HID device report handshanke error to lower layer, the event comes

enumerator **ESP_HIDD_GET_REPORT_EVT**

When HID device receives GET_REPORT request from host, the event comes

enumerator **ESP_HIDD_SET_REPORT_EVT**

When HID device receives SET_REPORT request from host, the event comes

enumerator **ESP_HIDD_SET_PROTOCOL_EVT**

When HID device receives SET_PROTOCOL request from host, the event comes

enumerator **ESP_HIDD_INTR_DATA_EVT**

When HID device receives DATA from host on intr, the event comes

enumerator **ESP_HIDD_VC_UNPLUG_EVT**

When HID device initiates Virtual Cable Unplug, the event comes

enumerator **ESP_HIDD_API_ERR_EVT**

When HID device has API error, the event comes

enum **esp_hidd_status_t**

Values:

enumerator **ESP_HIDD_SUCCESS**

enumerator **ESP_HIDD_ERROR**

general ESP HD error

enumerator **ESP_HIDD_NO_RES**

out of system resources

enumerator **ESP_HIDD_BUSY**

Temporarily can not handle this request.

enumerator **ESP_HIDD_NO_DATA**

No data.

enumerator **ESP_HIDD_NEED_INIT**

HIDD module shall init first

enumerator **ESP_HIDD_NEED_DEINIT**

HIDD module shall deinit first

enumerator **ESP_HIDD_NEED_REG**

HIDD module shall register first

enumerator **ESP_HIDD_NEED_DEREG**

HIDD module shall deregister first

enumerator **ESP_HIDD_NO_CONNECTION**

connection may have been closed

Bluetooth HID Host API

Overview A Bluetooth HID host is a device or software that is capable of connecting and communicating with Bluetooth HID devices, such as keyboards, mice. Users can use the Bluetooth HID Host APIs to send output data or control commands to the HID devices, enabling them to control the behavior or settings of the devices.

Application Example Check bluetooth folder in ESP-IDF examples, which contains the following application:

- Example bluetooth/esp_hid_host is implemented using the generic esp_hid APIs. esp_hid APIs are build upon the Bluetooth HID APIs and can be a reference.

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_hidh_api.h

Functions

esp_err_t **esp_bt_hid_host_register_callback**(*esp_hh_cb_t* callback)

This function is called to init callbacks with HID host module.

Parameters **callback** –**[in]** pointer to the init callback function.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_init**(void)

This function initializes HID host. This function should be called after esp_bluedroid_enable() and esp_bluedroid_init() success, and should be called after esp_bt_hid_host_register_callback(). When the operation is complete the callback function will be called with ESP_HIDH_INIT_EVT.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_deinit** (void)

Closes the interface. This function should be called after esp_bluedroid_enable() and esp_bluedroid_init() success, and should be called after esp_bt_hid_host_init(). When the operation is complete the callback function will be called with ESP_HIDH_DEINIT_EVT.

Returns - ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_connect** (*esp_bd_addr_t* bd_addr)

Connect to HID device. When the operation is complete the callback function will be called with ESP_HIDH_OPEN_EVT.

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.
Returns - ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_disconnect** (*esp_bd_addr_t* bd_addr)

Disconnect from HID device. When the operation is complete the callback function will be called with ESP_HIDH_CLOSE_EVT.

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.
Returns - ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_virtual_cable_unplug** (*esp_bd_addr_t* bd_addr)

Virtual UnPlug (VUP) the specified HID device. When the operation is complete the callback function will be called with ESP_HIDH_VC_UNPLUG_EVT.

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.
Returns - ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_set_info** (*esp_bd_addr_t* bd_addr, *esp_hidh_hid_info_t* *hid_info)

Set the HID device descriptor for the specified HID device. When the operation is complete the callback function will be called with ESP_HIDH_SET_INFO_EVT.

Parameters
- **bd_addr** –**[in]** Remote device bluetooth device address.
- **hid_info** –**[in]** HID device descriptor structure.

Returns - ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_get_protocol** (*esp_bd_addr_t* bd_addr)

Get the HID proto mode. When the operation is complete the callback function will be called with ESP_HIDH_GET_PROTO_EVT.

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.
Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_set_protocol** (*esp_bd_addr_t* bd_addr, *esp_hidh_protocol_mode_t* protocol_mode)

Set the HID proto mode. When the operation is complete the callback function will be called with ESP_HIDH_SET_PROTO_EVT.

Parameters
- **bd_addr** –**[in]** Remote device bluetooth device address.
- **protocol_mode** –**[in]** Protocol mode type.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_get_idle**(*esp_bd_addr_t* bd_addr)

Get the HID Idle Time. When the operation is complete the callback function will be called with ESP_HIDH_GET_IDLE_EVT.

Parameters **bd_addr** –**[in]** Remote device bluetooth device address.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_hid_host_set_idle**(*esp_bd_addr_t* bd_addr, uint16_t idle_time)

Set the HID Idle Time. When the operation is complete the callback function will be called with ESP_HIDH_SET_IDLE_EVT.

Parameters

- **bd_addr** –**[in]** Remote device bluetooth device address.
- **idle_time** –**[in]** Idle time rate

Returns - ESP_OK: success

- other: failed

esp_err_t **esp_bt_hid_host_get_report**(*esp_bd_addr_t* bd_addr, *esp_hidh_report_type_t* report_type, uint8_t report_id, int buffer_size)

Send a GET_REPORT to HID device. When the operation is complete the callback function will be called with ESP_HIDH_GET_RPT_EVT.

Parameters

- **bd_addr** –**[in]** Remote device bluetooth device address.
- **report_type** –**[in]** Report type
- **report_id** –**[in]** Report id
- **buffer_size** –**[in]** Buffer size

Returns - ESP_OK: success

- other: failed

esp_err_t **esp_bt_hid_host_set_report**(*esp_bd_addr_t* bd_addr, *esp_hidh_report_type_t* report_type, uint8_t *report, size_t len)

Send a SET_REPORT to HID device. When the operation is complete the callback function will be called with ESP_HIDH_SET_RPT_EVT.

Parameters

- **bd_addr** –**[in]** Remote device bluetooth device address.
- **report_type** –**[in]** Report type
- **report** –**[in]** Report data pointer
- **len** –**[in]** Report data length

Returns - ESP_OK: success

- other: failed

esp_err_t **esp_bt_hid_host_send_data**(*esp_bd_addr_t* bd_addr, uint8_t *data, size_t len)

Send data to HID device. When the operation is complete the callback function will be called with ESP_HIDH_DATA_EVT.

Parameters

- **bd_addr** –**[in]** Remote device bluetooth device address.
- **data** –**[in]** Data pointer
- **len** –**[in]** Data length

Returns - ESP_OK: success

- other: failed

Unions

union **esp_hidh_cb_param_t**

#include <esp_hidh_api.h> HID host callback parameters union.

Public Members

struct *esp_hidh_cb_param_t*::*hidh_init_evt_param* **init**
HIDH callback param of ESP_HIDH_INIT_EVT

struct *esp_hidh_cb_param_t*::*hidh_uninit_evt_param* **deinit**
HIDH callback param of ESP_HIDH_DEINIT_EVT

struct *esp_hidh_cb_param_t*::*hidh_open_evt_param* **open**
HIDH callback param of ESP_HIDH_OPEN_EVT

struct *esp_hidh_cb_param_t*::*hidh_close_evt_param* **close**
HIDH callback param of ESP_HIDH_CLOSE_EVT

struct *esp_hidh_cb_param_t*::*hidh_unplug_evt_param* **unplug**
HIDH callback param of ESP_HIDH_VC_UNPLUG_EVT

struct *esp_hidh_cb_param_t*::*hidh_get_proto_evt_param* **get_proto**
HIDH callback param of ESP_HIDH_GET_PROTO_EVT

struct *esp_hidh_cb_param_t*::*hidh_set_proto_evt_param* **set_proto**
HIDH callback param of ESP_HIDH_SET_PROTO_EVT

struct *esp_hidh_cb_param_t*::*hidh_get_rpt_evt_param* **get_rpt**
HIDH callback param of ESP_HIDH_GET_RPT_EVT

struct *esp_hidh_cb_param_t*::*hidh_set_rpt_evt_param* **set_rpt**
HIDH callback param of ESP_HIDH_SET_RPT_EVT

struct *esp_hidh_cb_param_t*::*hidh_send_data_evt_param* **send_data**
HIDH callback param of ESP_HIDH_DATA_EVT

struct *esp_hidh_cb_param_t*::*hidh_get_idle_evt_param* **get_idle**
HIDH callback param of ESP_HIDH_GET_IDLE_EVT

struct *esp_hidh_cb_param_t*::*hidh_set_idle_evt_param* **set_idle**
HIDH callback param of ESP_HIDH_SET_IDLE_EVT

struct *esp_hidh_cb_param_t*::*hidh_data_ind_evt_param* **data_ind**
HIDH callback param of ESP_HIDH_DATA_IND_EVT

struct *esp_hidh_cb_param_t*::*hidh_add_dev_evt_param* **add_dev**
HIDH callback param of ESP_HIDH_ADD_DEV_EVT

struct *esp_hidh_cb_param_t*::*hidh_rmv_dev_evt_param* **rmv_dev**
HIDH callback param of ESP_HIDH_RMV_DEV_EVT

struct *esp_hidh_cb_param_t*::*hidh_get_dscp_evt_param* **dscp**
HIDH callback param of ESP_HIDH_GET_DSCP_EVT

struct *esp_hidh_cb_param_t*::*hidh_set_info_evt_param* **set_info**

HIDH callback param of ESP_HIDH_SET_INFO_EVT

struct **hidh_add_dev_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_ADD_DEV_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

esp_bd_addr_t **bd_addr**

device address

struct **hidh_close_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_CLOSE_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **reason**

lower layer failed reason(ref hiddefs.h)

esp_hidh_connection_state_t **conn_status**

connection status

uint8_t **handle**

device handle

struct **hidh_data_ind_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_DATA_IND_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

esp_hidh_protocol_mode_t **proto_mode**
protocol mode

uint16_t **len**
data length

uint8_t ***data**
data pointer

struct **hidh_get_dscp_evt_param**
#include <esp_hidh_api.h> ESP_HIDH_GET_DSCP_EVT.

Public Members

esp_hidh_status_t **status**
operation status

uint8_t **handle**
device handle

bool **added**
Indicate if added

uint16_t **vendor_id**
Vendor ID

uint16_t **product_id**
Product ID

uint16_t **version**
Version

uint16_t **ssr_max_latency**
SSR max latency in slots

uint16_t **ssr_min_tout**
SSR min timeout in slots

uint8_t **ctry_code**
Country Code

uint16_t **dl_len**
Device descriptor length

uint8_t ***dsc_list**
Device descriptor pointer

struct **hidh_get_idle_evt_param**
#include <esp_hidh_api.h> ESP_HIDH_GET_IDLE_EVT.

Public Members

esp_hidh_status_t **status**
operation status

uint8_t **handle**
device handle

uint8_t **idle_rate**
idle rate

struct **hidh_get_proto_evt_param**
#include <*esp_hidh_api.h*> ESP_HIDH_GET_PROTO_EVT.

Public Members

esp_hidh_status_t **status**
operation status

uint8_t **handle**
device handle

esp_hidh_protocol_mode_t **proto_mode**
protocol mode

struct **hidh_get_rpt_evt_param**
#include <*esp_hidh_api.h*> ESP_HIDH_GET_RPT_EVT.

Public Members

esp_hidh_status_t **status**
operation status

uint8_t **handle**
device handle

uint16_t **len**
data length

uint8_t ***data**
data pointer

struct **hidh_init_evt_param**
#include <*esp_hidh_api.h*> ESP_HIDH_INIT_EVT.

Public Members

esp_hidh_status_t **status**

status

struct **hidh_open_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_OPEN_EVT.

Public Members

esp_hidh_status_t **status**

operation status

esp_hidh_connection_state_t **conn_status**

connection status

bool **is_orig**

indicate if host intiate the connection

uint8_t **handle**

device handle

esp_bd_addr_t **bd_addr**

device address

struct **hidh_rmv_dev_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_RMV_DEV_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

esp_bd_addr_t **bd_addr**

device address

struct **hidh_send_data_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_DATA_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

uint8_t **reason**

lower layer failed reason(ref hiddefs.h)

struct **hidh_set_idle_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_SET_IDLE_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

struct **hidh_set_info_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_SET_INFO_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

esp_bd_addr_t **bd_addr**

device address

struct **hidh_set_proto_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_SET_PROTO_EVT.

Public Members

esp_hidh_status_t **status**

operation status

uint8_t **handle**

device handle

struct **hidh_set_rpt_evt_param**

#include <esp_hidh_api.h> ESP_HIDH_SET_RPT_EVT.

Public Members

esp_hidh_status_t **status**
operation status

uint8_t **handle**
device handle

struct **hidh_uninit_evt_param**
#include <esp_hidh_api.h> ESP_HIDH_DEINIT_EVT.

Public Members

esp_hidh_status_t **status**
status

struct **hidh_unplug_evt_param**
#include <esp_hidh_api.h> ESP_HIDH_VC_UNPLUG_EVT.

Public Members

esp_hidh_status_t **status**
operation status

esp_hidh_connection_state_t **conn_status**
connection status

uint8_t **handle**
device handle

Structures

struct **esp_hidh_hid_info_t**
HID device information from HID Device Service Record and Device ID Service Record.

Public Members

int **attr_mask**
device attribute bit mask, refer to esp_hidh_dev_attr_t

uint8_t **sub_class**
HID device subclass

uint8_t **app_id**
application ID, refer to esp_hidh_dev_app_id_t

int **vendor_id**

Device ID information: vendor ID

int **product_id**

Device ID information: product ID

int **version**

Device ID information: version

uint8_t **ctry_code**

SDP attrbutes of HID devices: HID country code (https://www.usb.org/sites/default/files/hid1_11.pdf)

int **dl_len**

SDP attrbutes of HID devices: HID device descriptor length

uint8_t **dsc_list**[BTHH_MAX_DSC_LEN]

SDP attrbutes of HID devices: HID device descriptor definition

Macros

BTHH_MAX_DSC_LEN

maximum size of HID Device report descriptor

Type Definitions

typedef void (***esp_hh_cb_t**)(*esp_hidh_cb_event_t* event, *esp_hidh_cb_param_t* *param)

HID host callback function type.

Param event Event type
Param param Point to callback parameter, currently is union type

Enumerations

enum **esp_hidh_connection_state_t**

HID host connection state.

Values:

enumerator **ESP_HIDH_CONN_STATE_CONNECTED**

connected state

enumerator **ESP_HIDH_CONN_STATE_CONNECTING**

connecting state

enumerator **ESP_HIDH_CONN_STATE_DISCONNECTED**

disconnected state

enumerator **ESP_HIDH_CONN_STATE_DISCONNECTING**

disconnecting state

enumerator **ESP_HIDH_CONN_STATE_UNKNOWN**

unknown state (initial state)

enum **esp_hidh_status_t**

HID handshake error code and vendor-defined result code.

Values:

enumerator **ESP_HIDH_OK**

successful

enumerator **ESP_HIDH_HS_HID_NOT_READY**

handshake error: device not ready

enumerator **ESP_HIDH_HS_INVALID_RPT_ID**

handshake error: invalid report ID

enumerator **ESP_HIDH_HS_TRANS_NOT_SPT**

handshake error: HID device does not support the request

enumerator **ESP_HIDH_HS_INVALID_PARAM**

handshake error: parameter value does not meet the expected criteria of called function or API

enumerator **ESP_HIDH_HS_ERROR**

handshake error: HID device could not identify the error condition

enumerator **ESP_HIDH_ERR**

general ESP HID Host error

enumerator **ESP_HIDH_ERR_SDP**

SDP error

enumerator **ESP_HIDH_ERR_PROTO**

SET_PROTOCOL error, only used in ESP_HIDH_OPEN_EVT callback

enumerator **ESP_HIDH_ERR_DB_FULL**

device database full, used in ESP_HIDH_OPEN_EVT/ESP_HIDH_ADD_DEV_EVT

enumerator **ESP_HIDH_ERR_TOD_UNSPT**

type of device not supported

enumerator **ESP_HIDH_ERR_NO_RES**

out of system resources

enumerator **ESP_HIDH_ERR_AUTH_FAILED**

authentication fail

enumerator **ESP_HIDH_ERR_HDL**

connection handle error

enumerator **ESP_HIDH_ERR_SEC**

encryption error

enumerator **ESP_HIDH_BUSY**

vendor-defined: temporarily can not handle this request

enumerator **ESP_HIDH_NO_DATA**

vendor-defined: no data.

enumerator **ESP_HIDH_NEED_INIT**

vendor-defined: HIDH module shall initialize first

enumerator **ESP_HIDH_NEED_DEINIT**

vendor-defined: HIDH module shall de-deinitialize first

enumerator **ESP_HIDH_NO_CONNECTION**

vendor-defined: connection may have been closed

enum **esp_hidh_protocol_mode_t**

HID host protocol modes.

Values:

enumerator **ESP_HIDH_BOOT_MODE**

boot protocol mode

enumerator **ESP_HIDH_REPORT_MODE**

report protocol mode

enumerator **ESP_HIDH_UNSUPPORTED_MODE**

unsupported protocol mode

enum **esp_hidh_report_type_t**

HID host report types.

Values:

enumerator **ESP_HIDH_REPORT_TYPE_OTHER**

unsupported report type

enumerator **ESP_HIDH_REPORT_TYPE_INPUT**

input report type

enumerator **ESP_HIDH_REPORT_TYPE_OUTPUT**

output report type

enumerator **ESP_HIDH_REPORT_TYPE_FEATURE**

feature report type

enum **esp_hidh_cb_event_t**

HID host callback function events.

Values:

enumerator **ESP_HIDH_INIT_EVT**

when HID host is initialized, the event comes

enumerator **ESP_HIDH_DEINIT_EVT**

when HID host is deinitialized, the event comes

enumerator **ESP_HIDH_OPEN_EVT**

when HID host connection opened, the event comes

enumerator **ESP_HIDH_CLOSE_EVT**

when HID host connection closed, the event comes

enumerator **ESP_HIDH_GET_RPT_EVT**

when Get_Report command is called, the event comes

enumerator **ESP_HIDH_SET_RPT_EVT**

when Set_Report command is called, the event comes

enumerator **ESP_HIDH_GET_PROTO_EVT**

when Get_Protocol command is called, the event comes

enumerator **ESP_HIDH_SET_PROTO_EVT**

when Set_Protocol command is called, the event comes

enumerator **ESP_HIDH_GET_IDLE_EVT**

when Get_Idle command is called, the event comes

enumerator **ESP_HIDH_SET_IDLE_EVT**

when Set_Idle command is called, the event comes

enumerator **ESP_HIDH_GET_DSCP_EVT**

when HIDH is initialized, the event comes

enumerator **ESP_HIDH_ADD_DEV_EVT**

when a device is added, the event comes

enumerator **ESP_HIDH_RMV_DEV_EVT**

when a device is removed, the event comes

enumerator **ESP_HIDH_VC_UNPLUG_EVT**

when virtually unplugged, the event comes

enumerator **ESP_HIDH_DATA_EVT**

when send data on interrupt channel, the event comes

enumerator **ESP_HIDH_DATA_IND_EVT**

when receive data on interrupt channel, the event comes

enumerator **ESP_HIDH_SET_INFO_EVT**

when set the HID device descriptor, the event comes

enum **esp_hidh_dev_attr_t**

HID device information from HID Device Service Record and Device ID Service Record.

Values:

enumerator **ESP_HIDH_DEV_ATTR_VIRTUAL_CABLE**

whether Virtual Cables is supported

enumerator **ESP_HIDH_DEV_ATTR_NORMALLY_CONNECTABLE**

whether device is in Page Scan mode when there is no active connection

enumerator **ESP_HIDH_DEV_ATTR_RECONNECT_INITIATE**

whether the HID device inititates the reconnection process

enum **esp_hidh_dev_app_id_t**

application ID(non-zero) for each type of device

Values:

enumerator **ESP_HIDH_APP_ID_MOUSE**

pointing device

enumerator **ESP_HIDH_APP_ID_KEYBOARD**

keyboard

enumerator **ESP_HIDH_APP_ID_REMOTE_CONTROL**

remote control

enumerator **ESP_HIDH_APP_ID_JOYSTICK**

joystick

enumerator **ESP_HIDH_APP_ID_GAMEPAD**

gamepad

Classic Bluetooth L2CAP API

Application Example Check bluetooth/bluedroid/classic_bt folder in ESP-IDF examples, which contains the following application:

- This is a BT_L2CAP demo. This demo can connect, send and receive L2CAP data bluetooth/bluedroid/classic_bt/bt_l2cap_client, bluetooth/bluedroid/classic_bt/bt_l2cap_server

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_l2cap_bt_api.h

Functions

esp_err_t **esp_bt_l2cap_register_callback** (*esp_bt_l2cap_cb_t* callback)

This function is called to init callbacks with L2CAP module.

Parameters **callback** –**[in]** pointer to the init callback function.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_init** (void)

This function is called to init L2CAP module. When the operation is completed, the callback function will be called with ESP_BT_L2CAP_INIT_EVT. This function should be called after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_deinit** (void)

This function is called to uninit l2cap module. The operation will close all active L2CAP connection first, then the callback function will be called with ESP_BT_L2CAP_CLOSE_EVT, and the number of ESP_BT_L2CAP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback function will be called with ESP_BT_L2CAP_UNINIT_EVT. This function should be called after esp_bt_l2cap_init() completes successfully.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_connect** (*esp_bt_l2cap_cntl_flags_t* cntl_flag, uint16_t remote_psm, *esp_bd_addr_t* peer_bd_addr)

This function makes an L2CAP connection to a remote BD Address. When the connection is initiated or failed to initiate, the callback is called with ESP_BT_L2CAP_CL_INIT_EVT. When the connection is established or failed, the callback is called with ESP_BT_L2CAP_OPEN_EVT. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Parameters

- **cntl_flag** –**[in]** Lower 16-bit security settings mask.
- **remote_psm** –**[in]** Remote device bluetooth Profile PSM.
- **peer_bd_addr** –**[in]** Remote device bluetooth device address.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_start_srv** (*esp_bt_l2cap_cntl_flags_t* cntl_flag, uint16_t local_psm)

This function create a L2CAP server and starts listening for an L2CAP connection request from a remote Bluetooth device. When the server is started successfully, the callback is called with ESP_BT_L2CAP_START_EVT. When the connection is established, the callback is called with ESP_BT_L2CAP_OPEN_EVT. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Parameters

- **cntl_flag** –**[in]** Lower 16-bit security settings mask.
- **local_psm** –**[in]** Dynamic PSM.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_stop_all_srv**(void)

This function stops all L2CAP servers. The operation will close all active L2CAP connection first, then the callback function will be called with ESP_BT_L2CAP_CLOSE_EVT, and the number of ESP_BT_L2CAP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_BT_L2CAP_SRV_STOP_EVT. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_stop_srv**(uint16_t local_psm)

This function stops a specific L2CAP server. The operation will close all active L2CAP connection first on the specific L2CAP server, then the callback function will be called with ESP_BT_L2CAP_CLOSE_EVT, and the number of ESP_BT_L2CAP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_BT_L2CAP_SRV_STOP_EVT. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Parameters **local_psm** – **[in]** Dynamic PSM.

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_vfs_register**(void)

This function is used to register VFS. Only supports write, read and close. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Returns
- ESP_OK: success
- other: failed

esp_err_t **esp_bt_l2cap_vfs_unregister**(void)

This function is used to unregister VFS. This function must be called after esp_bt_l2cap_init() successful and before esp_bt_l2cap_deinit().

Returns
- ESP_OK: success
- other: failed

Unions

union **esp_bt_l2cap_cb_param_t**

#include <esp_l2cap_bt_api.h> L2CAP callback parameters union.

Public Members

struct *esp_bt_l2cap_cb_param_t*::*l2cap_init_evt_param* **init**

L2CAP callback param of ESP_BT_L2CAP_INIT_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_uninit_evt_param* **uninit**

L2CAP callback param of ESP_BT_L2CAP_UNINIT_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_open_evt_param* **open**

L2CAP callback param of ESP_BT_L2CAP_OPEN_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_close_evt_param* **close**

L2CAP callback param of ESP_BT_L2CAP_CLOSE_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_start_evt_param* **start**

L2CAP callback param of ESP_BT_L2CAP_START_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_cl_init_evt_param* **cl_init**

L2CAP callback param of ESP_BT_L2CAP_CL_INIT_EVT

struct *esp_bt_l2cap_cb_param_t*::*l2cap_srv_stop_evt_param* **srv_stop**

L2CAP callback param of ESP_BT_L2CAP_SRV_STOP_EVT

struct **l2cap_cl_init_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_CL_INIT_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

uint32_t **handle**

The connection handle

uint8_t **sec_id**

security ID used by this server

struct **l2cap_close_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_CLOSE_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

uint32_t **handle**

The connection handle

bool **async**

FALSE, if local initiates disconnect

struct **l2cap_init_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_INIT_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

struct **l2cap_open_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_OPEN_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

uint32_t **handle**

The connection handle

int **fd**

File descriptor

esp_bd_addr_t **rem_bda**

The peer address

int32_t **tx_mtu**

The transmit MTU

struct **l2cap_srv_stop_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_SRV_STOP_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

uint8_t **psm**

local psm

struct **l2cap_start_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_START_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

uint32_t **handle**

The connection handle

uint8_t **sec_id**

security ID used by this server

struct **l2cap_uninit_evt_param**

#include <esp_l2cap_bt_api.h> ESP_BT_L2CAP_UNINIT_EVT.

Public Members

esp_bt_l2cap_status_t **status**

status

Macros

ESP_BT_L2CAP_SEC_NONE

Security Setting Mask. Use these three mask mode:

a. ESP_BT_L2CAP_SEC_NONE
b. ESP_BT_L2CAP_SEC_AUTHENTICATE
c. (ESP_BT_L2CAP_SEC_ENCRYPT|ESP_BT_L2CAP_SEC_AUTHENTICATE) No security

ESP_BT_L2CAP_SEC_AUTHORIZE

Authorization required

ESP_BT_L2CAP_SEC_AUTHENTICATE

Authentication required

ESP_BT_L2CAP_SEC_ENCRYPT

Encryption required

Type Definitions

typedef uint32_t **esp_bt_l2cap_cntl_flags_t**

typedef void (***esp_bt_l2cap_cb_t**)(*esp_bt_l2cap_cb_event_t* event, *esp_bt_l2cap_cb_param_t* *param)

L2CAP callback function type.

Param event Event type
Param param Point to callback parameter, currently is union type

Enumerations

enum **esp_bt_l2cap_status_t**

L2CAP operation success and failure codes.

Values:

enumerator **ESP_BT_L2CAP_SUCCESS**

Successful operation.

enumerator **ESP_BT_L2CAP_FAILURE**

Generic failure.

enumerator **ESP_BT_L2CAP_BUSY**

Temporarily can not handle this request.

enumerator **ESP_BT_L2CAP_NO_RESOURCE**

No more resource

enumerator **ESP_BT_L2CAP_NEED_INIT**

L2CAP module shall init first

enumerator **ESP_BT_L2CAP_NEED_DEINIT**

L2CAP module shall deinit first

enumerator **ESP_BT_L2CAP_NO_CONNECTION**

Connection may have been closed

enumerator **ESP_BT_L2CAP_NO_SERVER**

No server

enum **esp_bt_l2cap_cb_event_t**

L2CAP callback function events.

Values:

enumerator **ESP_BT_L2CAP_INIT_EVT**

When L2CAP is initialized, the event comes

enumerator **ESP_BT_L2CAP_UNINIT_EVT**

When L2CAP is deinitialized, the event comes

enumerator **ESP_BT_L2CAP_OPEN_EVT**

When L2CAP Client connection open, the event comes

enumerator **ESP_BT_L2CAP_CLOSE_EVT**

When L2CAP connection closed, the event comes

enumerator **ESP_BT_L2CAP_START_EVT**

When L2CAP server started, the event comes

enumerator **ESP_BT_L2CAP_CL_INIT_EVT**

When L2CAP client initiated a connection, the event comes

enumerator **ESP_BT_L2CAP_SRV_STOP_EVT**

When L2CAP server stopped, the event comes

BT SDP APIs

Overview Bluetooth SDP reference APIs.

API Reference

Header File

- components/bt/host/bluedroid/api/include/api/esp_sdp_api.h

Functions

esp_err_t **esp_sdp_register_callback**(*esp_sdp_cb_t* callback)

This function is called to init callbacks with SDP module.

Parameters **callback** –**[in]** pointer to the init callback function.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_sdp_init**(void)

This function is called to init SDP module. When the operation is completed, the callback function will be called with ESP_SDP_INIT_EVT. This function should be called after esp_bluedroid_enable() completes successfully.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_sdp_deinit**(void)

This function is called to de-initialize SDP module. The operation will remove all SDP records, then the callback function will be called with ESP_SDP_REMOVE_RECORD_COMP_EVT, and the number of ESP_SDP_REMOVE_RECORD_COMP_EVT is equal to the number of SDP records.When the operation is completed, the callback function will be called with ESP_SDP_DEINIT_EVT. This function should be called after esp_sdp_init() completes successfully.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_sdp_search_record**(*esp_bd_addr_t* bd_addr, *esp_bt_uuid_t* uuid)

This function is called to performs service discovery for the services provided by the given peer device. When the operation is completed, the callback function will be called with ESP_SDP_SEARCH_COMP_EVT. This function must be called after esp_sdp_init() successful and before esp_sdp_deinit().

Parameters

- **bd_addr** –**[in]** Remote device bluetooth device address.
- **uuid** –**[in]** Service UUID of the remote device.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_sdp_create_record**(*esp_bluetooth_sdp_record_t* *record)

This function is called to create SDP records. When the operation is completed, the callback function will be called with ESP_SDP_CREATE_RECORD_COMP_EVT. This function must be called after esp_sdp_init() successful and before esp_sdp_deinit().

Parameters **record** –**[in]** The SDP record to create.

Returns

- ESP_OK: success
- other: failed

esp_err_t **esp_sdp_remove_record**(int record_handle)

This function is called to remove a SDP record. When the operation is completed, the callback function will be called with ESP_SDP_REMOVE_RECORD_COMP_EVT. This function must be called after esp_sdp_init() successful and before esp_sdp_deinit().

Parameters **record_handle** –**[in]** The SDP record handle.
Returns
- ESP_OK: success
- other: failed

Unions

union **esp_bluetooth_sdp_record_t**

#include <esp_sdp_api.h> SDP record parameters union.

Public Members

esp_bluetooth_sdp_hdr_overlay_t **hdr**

General info

esp_bluetooth_sdp_mas_record_t **mas**

Message Access Profile - Server

esp_bluetooth_sdp_mns_record_t **mns**

Message Access Profile - Client (Notification Server)

esp_bluetooth_sdp_pse_record_t **pse**

Phone Book Profile - Server

esp_bluetooth_sdp_pce_record_t **pce**

Phone Book Profile - Client

esp_bluetooth_sdp_ops_record_t **ops**

Object Push Profile

esp_bluetooth_sdp_sap_record_t **sap**

SIM Access Profile

union **esp_sdp_cb_param_t**

#include <esp_sdp_api.h> SDP callback parameters union.

Public Members

struct *esp_sdp_cb_param_t*::*sdp_init_evt_param* **init**

SDP callback param of ESP_SDP_INIT_EVT

struct *esp_sdp_cb_param_t*::*sdp_deinit_evt_param* **deinit**

SDP callback param of ESP_SDP_DEINIT_EVT

struct *esp_sdp_cb_param_t*::*sdp_search_evt_param* **search**

SDP callback param of ESP_SDP_SEARCH_COMP_EVT

struct *esp_sdp_cb_param_t*::*sdp_crate_record_evt_param* **create_record**

SDP callback param of ESP_SDP_CREATE_RECORD_COMP_EVT

struct *esp_sdp_cb_param_t*::*sdp_remove_record_evt_param* **remove_record**

SDP callback param of ESP_SDP_REMOVE_RECORD_COMP_EVT

struct **sdp_crate_record_evt_param**

#include <*esp_sdp_api.h*> ESP_SDP_CREATE_RECORD_COMP_EVT.

Public Members

esp_sdp_status_t **status**

status

int **record_handle**

SDP record handle

struct **sdp_deinit_evt_param**

#include <*esp_sdp_api.h*> ESP_SDP_DEINIT_EVT.

Public Members

esp_sdp_status_t **status**

status

struct **sdp_init_evt_param**

#include <*esp_sdp_api.h*> ESP_SDP_INIT_EVT.

Public Members

esp_sdp_status_t **status**

status

struct **sdp_remove_record_evt_param**

#include <*esp_sdp_api.h*> ESP_SDP_REMOVE_RECORD_COMP_EVT.

Public Members

esp_sdp_status_t **status**

status

struct **sdp_search_evt_param**

#include <*esp_sdp_api.h*> ESP_SDP_SEARCH_COMP_EVT.

Public Members

esp_sdp_status_t **status**

status

esp_bd_addr_t **remote_addr**

remote device address

esp_bt_uuid_t **sdp_uuid**

service uuid

int **record_count**

Number of SDP records

esp_bluetooth_sdp_record_t ***records**

SDP records

Structures

struct **bluetooth_sdp_hdr_overlay**

Some signals need additional pointers, hence we introduce a generic way to handle these pointers.

Public Members

esp_bluetooth_sdp_types_t **type**

SDP type

esp_bt_uuid_t **uuid**

UUID type，include uuid and uuid length

uint32_t **service_name_length**

Service name length

char ***service_name**

service name

int32_t **rfcomm_channel_number**

rfcomm channel number, if not used set to -1

int32_t **l2cap_psm**

l2cap psm, if not used set to -1

int32_t **profile_version**

profile version

int **user1_ptr_len**

see esp_bluetooth_sdp_ops_record_t

uint8_t ***user1_ptr**

see esp_bluetooth_sdp_ops_record_t

int **user2_ptr_len**

see esp_bluetooth_sdp_ops_record_t

uint8_t *`user2_ptr`

see esp_bluetooth_sdp_ops_record_t

struct `bluetooth_sdp_mas_record`

Message Access Profile - Server parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t `hdr`

General info

uint32_t `mas_instance_id`

MAS Instance ID

uint32_t `supported_features`

Map supported features

uint32_t `supported_message_types`

Supported message types

struct `bluetooth_sdp_mns_record`

Message Access Profile - Client (Notification Server) parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t `hdr`

General info

uint32_t `supported_features`

Supported features

struct `bluetooth_sdp_pse_record`

Phone Book Profile - Server parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t `hdr`

General info

uint32_t `supported_features`

Pbap Supported Features

uint32_t `supported_repositories`

Supported Repositories

struct `bluetooth_sdp_pce_record`

Phone Book Profile - Client parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t **hdr**

General info

struct **bluetooth_sdp_ops_record**

Object Push Profile parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t **hdr**

General info

int **supported_formats_list_len**

Supported formats list length

uint8_t **supported_formats_list**[SDP_OPP_SUPPORTED_FORMATS_MAX_LENGTH]

Supported formats list

struct **bluetooth_sdp_sap_record**

SIM Access Profile parameters.

Public Members

esp_bluetooth_sdp_hdr_overlay_t **hdr**

General info

Macros

ESP_SDP_SERVER_NAME_MAX

Service name max length

SDP_OPP_SUPPORTED_FORMATS_MAX_LENGTH

OPP supported format list maximum length

Type Definitions

typedef struct *bluetooth_sdp_hdr_overlay* **esp_bluetooth_sdp_hdr_overlay_t**

Some signals need additional pointers, hence we introduce a generic way to handle these pointers.

typedef struct *bluetooth_sdp_mas_record* **esp_bluetooth_sdp_mas_record_t**

Message Access Profile - Server parameters.

typedef struct *bluetooth_sdp_mns_record* **esp_bluetooth_sdp_mns_record_t**

Message Access Profile - Client (Notification Server) parameters.

typedef struct *bluetooth_sdp_pse_record* **esp_bluetooth_sdp_pse_record_t**

Phone Book Profile - Server parameters.

typedef struct *bluetooth_sdp_pce_record* **esp_bluetooth_sdp_pce_record_t**

Phone Book Profile - Client parameters.

typedef struct *bluetooth_sdp_ops_record* **esp_bluetooth_sdp_ops_record_t**

Object Push Profile parameters.

typedef struct *bluetooth_sdp_sap_record* **esp_bluetooth_sdp_sap_record_t**

SIM Access Profile parameters.

typedef void (***esp_sdp_cb_t**)(*esp_sdp_cb_event_t* event, *esp_sdp_cb_param_t* *param)

SDP callback function type.

Param event Event type
Param param Point to callback parameter, currently is union type

Enumerations

enum **esp_sdp_status_t**

Values:

enumerator **ESP_SDP_SUCCESS**

Successful operation.

enumerator **ESP_SDP_FAILURE**

Generic failure.

enumerator **ESP_SDP_NO_RESOURCE**

No more resource

enumerator **ESP_SDP_NEED_INIT**

SDP module shall init first

enumerator **ESP_SDP_NEED_DEINIT**

SDP module shall deinit first

enumerator **ESP_SDP_NO_CREATE_RECORD**

No record created

enum **esp_sdp_cb_event_t**

SDP callback function events.

Values:

enumerator **ESP_SDP_INIT_EVT**

When SDP is initialized, the event comes

enumerator **ESP_SDP_DEINIT_EVT**

When SDP is deinitialized, the event comes

enumerator **ESP_SDP_SEARCH_COMP_EVT**

When SDP search complete, the event comes

enumerator **ESP_SDP_CREATE_RECORD_COMP_EVT**

When create SDP records complete, the event comes

enumerator **ESP_SDP_REMOVE_RECORD_COMP_EVT**

When remove a SDP record complete, the event comes

enum **esp_bluetooth_sdp_types_t**

SDP record type.

Values:

enumerator **ESP_SDP_TYPE_RAW**

Used to carry raw SDP search data for unknown UUIDs

enumerator **ESP_SDP_TYPE_MAP_MAS**

Message Access Profile - Server

enumerator **ESP_SDP_TYPE_MAP_MNS**

Message Access Profile - Client (Notification Server)

enumerator **ESP_SDP_TYPE_PBAP_PSE**

Phone Book Profile - Server

enumerator **ESP_SDP_TYPE_PBAP_PCE**

Phone Book Profile - Client

enumerator **ESP_SDP_TYPE_OPP_SERVER**

Object Push Profile

enumerator **ESP_SDP_TYPE_SAP_SERVER**

SIM Access Profile

2.3.4 Controller && VHCI

Application Example

Check bluetooth/hci folder in ESP-IDF examples, which contains the following application:

- This is a BLE advertising demo with virtual HCI interface. Send Reset/ADV_PARAM/ADV_DATA/ADV_ENABLE HCI command for BLE advertising - bluetooth/hci/controller_vhci_ble_adv.

API Reference

Header File

- components/bt/include/esp32/include/esp_bt.h

Functions

esp_err_t **esp_ble_tx_power_set** (*esp_ble_power_type_t* power_type, *esp_power_level_t* power_level)

Set BLE TX power Connection Tx power should only be set after connection created.

Parameters
- **power_type** –: The type of which tx power, could set Advertising/Connection/Default and etc
- **power_level** –Power level(index) corresponding to absolute value(dbm)

Returns ESP_OK - success, other - failed

esp_power_level_t **esp_ble_tx_power_get** (*esp_ble_power_type_t* power_type)

Get BLE TX power Connection Tx power should only be get after connection created.

Parameters **power_type** –: The type of which tx power, could set Advertising/Connection/Default and etc

Returns >= 0 - Power level, < 0 - Invalid

esp_err_t **esp_bredr_tx_power_set** (*esp_power_level_t* min_power_level, *esp_power_level_t* max_power_level)

Set BR/EDR TX power BR/EDR power control will use the power in range of minimum value and maximum value. The power level will effect the global BR/EDR TX power, such inquire, page, connection and so on. Please call the function after esp_bt_controller_enable and before any function which cause RF do TX. So you can call the function before doing discovery, profile init and so on. For example, if you want BR/EDR use the new TX power to do inquire, you should call this function before inquire. Another word, If call this function when BR/EDR is in inquire(ING), please do inquire again after call this function. Default minimum power level is ESP_PWR_LVL_N0, and maximum power level is ESP_PWR_LVL_P3.

Parameters
- **min_power_level** –The minimum power level
- **max_power_level** –The maximum power level

Returns ESP_OK - success, other - failed

esp_err_t **esp_bredr_tx_power_get** (*esp_power_level_t* *min_power_level, *esp_power_level_t* *max_power_level)

Get BR/EDR TX power If the argument is not NULL, then store the corresponding value.

Parameters
- **min_power_level** –The minimum power level
- **max_power_level** –The maximum power level

Returns ESP_OK - success, other - failed

esp_err_t **esp_bredr_sco_datapath_set** (*esp_sco_data_path_t* data_path)

Set default SCO data path Should be called after controller is enabled, and before (e)SCO link is established.

Parameters **data_path** –SCO data path

Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_controller_init** (*esp_bt_controller_config_t* *cfg)

Initialize BT controller to allocate task and other resource. This function should be called only once, before any other BT functions are called.

Parameters **cfg** –Initial configuration of BT controller. Different from previous version, there's a mode and some connection configuration in "cfg" to configure controller work mode and allocate the resource which is needed.

Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_controller_deinit** (void)

De-initialize BT controller to free resource and delete task. You should stop advertising and scanning, as well as disconnect all existing connections before de-initializing BT controller.

This function should be called only once, after any other BT functions are called.

Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_controller_enable**(*esp_bt_mode_t* mode)

Enable BT controller. Due to a known issue, you cannot call esp_bt_controller_enable() a second time to change the controller mode dynamically. To change controller mode, call esp_bt_controller_disable() and then call esp_bt_controller_enable() with the new mode.

Parameters **mode** –: the mode(BLE/BT/BTDM) to enable. For compatible of API, retain this argument. This mode must be equal as the mode in "cfg" of esp_bt_controller_init().
Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_controller_disable**(void)

Disable BT controller.

Returns ESP_OK - success, other - failed

esp_bt_controller_status_t **esp_bt_controller_get_status**(void)

Get BT controller is initialised/de-initialised/enabled/disabled.

Returns status value

bool **esp_vhci_host_check_send_available**(void)

esp_vhci_host_check_send_available used for check actively if the host can send packet to controller or not.

Returns true for ready to send, false means cannot send packet

void **esp_vhci_host_send_packet**(uint8_t *data, uint16_t len)

esp_vhci_host_send_packet host send packet to controller

Should not call this function from within a critical section or when the scheduler is suspended.

Parameters
- **data** –the packet point
- **len** –the packet length

esp_err_t **esp_vhci_host_register_callback**(const *esp_vhci_host_callback_t* *callback)

esp_vhci_host_register_callback register the vhci reference callback struct defined by vhci_host_callback structure.

Parameters **callback** –*esp_vhci_host_callback* type variable
Returns ESP_OK - success, ESP_FAIL - failed

esp_err_t **esp_bt_controller_mem_release**(*esp_bt_mode_t* mode)

esp_bt_controller_mem_release release the controller memory as per the mode

This function releases the BSS, data and other sections of the controller to heap. The total size is about 70k bytes.

esp_bt_controller_mem_release(mode) should be called only before esp_bt_controller_init() or after esp_bt_controller_deinit().

Note that once BT controller memory is released, the process cannot be reversed. It means you cannot use the bluetooth mode which you have released by this function.

If your firmware will later upgrade the Bluetooth controller mode (BLE -> BT Classic or disabled -> enabled) then do not call this function.

If the app calls esp_bt_controller_enable(ESP_BT_MODE_BLE) to use BLE only then it is safe to call esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT) at initialization time to free unused BT Classic memory.

If the mode is ESP_BT_MODE_BTDM, then it may be useful to call API esp_bt_mem_release(ESP_BT_MODE_BTDM) instead, which internally calls esp_bt_controller_mem_release(ESP_BT_MODE_BTDM) and additionally releases the BSS and data consumed by the BT/BLE host stack to heap. For more details about usage please refer to the documentation of esp_bt_mem_release() function

Parameters **mode** –: the mode want to release memory

Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_mem_release**(*esp_bt_mode_t* mode)

esp_bt_mem_release release controller memory and BSS and data section of the BT/BLE host stack as per the mode

This function first releases controller memory by internally calling esp_bt_controller_mem_release(). Additionally, if the mode is set to ESP_BT_MODE_BTDM, it also releases the BSS and data consumed by the BT/BLE host stack to heap

Note that once BT memory is released, the process cannot be reversed. It means you cannot use the bluetooth mode which you have released by this function.

If your firmware will later upgrade the Bluetooth controller mode (BLE -> BT Classic or disabled -> enabled) then do not call this function.

If you never intend to use bluetooth in a current boot-up cycle, you can call esp_bt_mem_release(ESP_BT_MODE_BTDM) before esp_bt_controller_init or after esp_bt_controller_deinit.

For example, if a user only uses bluetooth for setting the WiFi configuration, and does not use bluetooth in the rest of the product operation" . In such cases, after receiving the WiFi configuration, you can disable/deinit bluetooth and release its memory. Below is the sequence of APIs to be called for such scenarios:

```
esp_bluedroid_disable();
esp_bluedroid_deinit();
esp_bt_controller_disable();
esp_bt_controller_deinit();
esp_bt_mem_release(ESP_BT_MODE_BTDM);
```

Note: In case of NimBLE host, to release BSS and data memory to heap, the mode needs to be set to ESP_BT_MODE_BTDM as controller is dual mode.

Parameters **mode** –: the mode whose memory is to be released
Returns ESP_OK - success, other - failed

esp_err_t **esp_bt_sleep_enable**(void)

enable bluetooth to enter modem sleep

Note that this function shall not be invoked before esp_bt_controller_enable()

There are currently two options for bluetooth modem sleep, one is ORIG mode, and another is EVED Mode. EVED Mode is intended for BLE only.

For ORIG mode: Bluetooth modem sleep is enabled in controller start up by default if CONFIG_CTRL_BTDM_MODEM_SLEEP is set and "ORIG mode" is selected. In ORIG modem sleep mode, bluetooth controller will switch off some components and pause to work every now and then, if there is no event to process; and wakeup according to the scheduled interval and resume the work. It can also wakeup earlier upon external request using function "esp_bt_controller_wakeup_request" .

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_bt_sleep_disable**(void)

disable bluetooth modem sleep

Note that this function shall not be invoked before esp_bt_controller_enable()

If esp_bt_sleep_disable() is called, bluetooth controller will not be allowed to enter modem sleep;

If ORIG modem sleep mode is in use, if this function is called, bluetooth controller may not immediately wake up if it is dormant then. In this case, esp_bt_controller_wakeup_request() can be used to shorten the time for wakeup.

Returns

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_scan_dupilcate_list_flush**(void)

Manually clear scan duplicate list.

Note that scan duplicate list will be automatically cleared when the maximum amount of device in the filter is reached the amount of device in the filter can be configured in menuconfig.

Note: This function name is incorrectly spelled, it will be fixed in release 5.x version.

Returns

- ESP_OK : success
- other : failed

void **esp_wifi_bt_power_domain_on**(void)

bt Wi-Fi power domain power on

void **esp_wifi_bt_power_domain_off**(void)

bt Wi-Fi power domain power off

Structures

struct **esp_bt_controller_config_t**

Controller config options, depend on config mask. Config mask indicate which functions enabled, this means some options or parameters of some functions enabled by config mask.

Public Members

uint16_t **controller_task_stack_size**

Bluetooth controller task stack size

uint8_t **controller_task_prio**

Bluetooth controller task priority

uint8_t **hci_uart_no**

If use UART1/2 as HCI IO interface, indicate UART number

uint32_t **hci_uart_baudrate**

If use UART1/2 as HCI IO interface, indicate UART baudrate

uint8_t **scan_duplicate_mode**

scan duplicate mode

uint8_t **scan_duplicate_type**

scan duplicate type

uint16_t **normal_adv_size**

Normal adv size for scan duplicate

uint16_t **mesh_adv_size**

Mesh adv size for scan duplicate

uint16_t **send_adv_reserved_size**

Controller minimum memory value

uint32_t **controller_debug_flag**

Controller debug log flag

uint8_t **mode**

Controller mode: BR/EDR, BLE or Dual Mode

uint8_t **ble_max_conn**

BLE maximum connection numbers

uint8_t **bt_max_acl_conn**

BR/EDR maximum ACL connection numbers

uint8_t **bt_sco_datapath**

SCO data path, i.e. HCI or PCM module

bool **auto_latency**

BLE auto latency, used to enhance classic BT performance

bool **bt_legacy_auth_vs_evt**

BR/EDR Legacy auth complete event required to protect from BIAS attack

uint8_t **bt_max_sync_conn**

BR/EDR maximum ACL connection numbers. Effective in menuconfig

uint8_t **ble_sca**

BLE low power crystal accuracy index

uint8_t **pcm_role**

PCM role (master & slave)

uint8_t **pcm_polar**

PCM polar trig (falling clk edge & rising clk edge)

bool **hli**

Using high level interrupt or not

uint16_t **dup_list_refresh_period**

Duplicate scan list refresh period

bool **ble_scan_backoff**

BLE scan backoff

uint32_t **magic**

Magic number

struct **esp_vhci_host_callback**

esp_vhci_host_callback used for vhci call host function to notify what host need to do

Public Members

void (***notify_host_send_available**)(void)

callback used to notify that the host can send packet to controller

int (***notify_host_recv**)(uint8_t *data, uint16_t len)

callback used to notify that the controller has a packet to send to the host

Macros

ESP_BT_CONTROLLER_CONFIG_MAGIC_VAL

BT_CONTROLLER_INIT_CONFIG_DEFAULT( )

Type Definitions

typedef struct *esp_vhci_host_callback* **esp_vhci_host_callback_t**

esp_vhci_host_callback used for vhci call host function to notify what host need to do

Enumerations

enum **esp_bt_mode_t**

Bluetooth mode for controller enable/disable.

Values:

enumerator **ESP_BT_MODE_IDLE**

Bluetooth is not running

enumerator **ESP_BT_MODE_BLE**

Run BLE mode

enumerator **ESP_BT_MODE_CLASSIC_BT**

Run Classic BT mode

enumerator **ESP_BT_MODE_BTDM**

Run dual mode

enum **[anonymous]**

BLE sleep clock accuracy(SCA), values for ble_sca field in *esp_bt_controller_config_t*, currently only ESP_BLE_SCA_500PPM and ESP_BLE_SCA_250PPM are supported.

Values:

enumerator **ESP_BLE_SCA_500PPM**

BLE SCA at 500ppm

enumerator **ESP_BLE_SCA_250PPM**

BLE SCA at 250ppm

enumerator **ESP_BLE_SCA_150PPM**

BLE SCA at 150ppm

enumerator **ESP_BLE_SCA_100PPM**

BLE SCA at 100ppm

enumerator **ESP_BLE_SCA_75PPM**

BLE SCA at 75ppm

enumerator **ESP_BLE_SCA_50PPM**

BLE SCA at 50ppm

enumerator **ESP_BLE_SCA_30PPM**

BLE SCA at 30ppm

enumerator **ESP_BLE_SCA_20PPM**

BLE SCA at 20ppm

enum **esp_bt_controller_status_t**

Bluetooth controller enable/disable/initialised/de-initialised status.

Values:

enumerator **ESP_BT_CONTROLLER_STATUS_IDLE**

enumerator **ESP_BT_CONTROLLER_STATUS_INITED**

enumerator **ESP_BT_CONTROLLER_STATUS_ENABLED**

enumerator **ESP_BT_CONTROLLER_STATUS_NUM**

enum **esp_ble_power_type_t**

BLE tx power type ESP_BLE_PWR_TYPE_CONN_HDL0-8: for each connection, and only be set after connection completed. when disconnect, the correspond TX power is not effected. ESP_BLE_PWR_TYPE_ADV : for advertising/scan response. ESP_BLE_PWR_TYPE_SCAN : for scan. ESP_BLE_PWR_TYPE_DEFAULT : if each connection's TX power is not set, it will use this default value. if neither in scan mode nor in adv mode, it will use this default value. If none of power type is set, system will use ESP_PWR_LVL_P3 as default for ADV/SCAN/CONN0-9.

Values:

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL0**

For connection handle 0

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL1**

For connection handle 1

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL2**

For connection handle 2

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL3**

For connection handle 3

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL4**

For connection handle 4

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL5**

For connection handle 5

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL6**

For connection handle 6

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL7**

For connection handle 7

enumerator **ESP_BLE_PWR_TYPE_CONN_HDL8**

For connection handle 8

enumerator **ESP_BLE_PWR_TYPE_ADV**

For advertising

enumerator **ESP_BLE_PWR_TYPE_SCAN**

For scan

enumerator **ESP_BLE_PWR_TYPE_DEFAULT**

For default, if not set other, it will use default value

enumerator **ESP_BLE_PWR_TYPE_NUM**

TYPE numbers

enum **esp_power_level_t**

Bluetooth TX power level(index), it’ s just a index corresponding to power(dbm).

Values:

enumerator **ESP_PWR_LVL_N12**

Corresponding to -12dbm

enumerator **ESP_PWR_LVL_N9**

Corresponding to -9dbm

enumerator **ESP_PWR_LVL_N6**

Corresponding to -6dbm

enumerator **ESP_PWR_LVL_N3**

Corresponding to -3dbm

enumerator **ESP_PWR_LVL_N0**

Corresponding to 0dbm

enumerator **ESP_PWR_LVL_P3**

Corresponding to +3dbm

enumerator **ESP_PWR_LVL_P6**

Corresponding to +6dbm

enumerator **ESP_PWR_LVL_P9**

Corresponding to +9dbm

enumerator **ESP_PWR_LVL_N14**

Backward compatibility! Setting to -14dbm will actually result to -12dbm

enumerator **ESP_PWR_LVL_N11**

Backward compatibility! Setting to -11dbm will actually result to -9dbm

enumerator **ESP_PWR_LVL_N8**

Backward compatibility! Setting to -8dbm will actually result to -6dbm

enumerator **ESP_PWR_LVL_N5**

Backward compatibility! Setting to -5dbm will actually result to -3dbm

enumerator **ESP_PWR_LVL_N2**

Backward compatibility! Setting to -2dbm will actually result to 0dbm

enumerator **ESP_PWR_LVL_P1**

Backward compatibility! Setting to +1dbm will actually result to +3dbm

enumerator **ESP_PWR_LVL_P4**

Backward compatibility! Setting to +4dbm will actually result to +6dbm

enumerator **ESP_PWR_LVL_P7**

Backward compatibility! Setting to +7dbm will actually result to +9dbm

enum **esp_sco_data_path_t**

Bluetooth audio data transport path.

Values:

enumerator **ESP_SCO_DATA_PATH_HCI**

data over HCI transport

enumerator **ESP_SCO_DATA_PATH_PCM**

data over PCM interface

2.3.5 ESP-BLE-MESH

With various features of ESP-BLE-MESH, users can create a managed flooding mesh network for several scenarios, such as lighting, sensor and etc.

For an ESP32 to join and work on a ESP-BLE-MESH network, it must be provisioned firstly. By provisioning, the ESP32, as an unprovisioned device, will join the ESP-BLE-MESH network and become a ESP-BLE-MESH node, communicating with other nodes within or beyond the radio range.

Apart from ESP-BLE-MESH nodes, inside ESP-BLE-MESH network, there is also ESP32 that works as ESP-BLE-MESH Provisioner, which could provision unprovisioned devices into ESP-BLE-MESH nodes and configure the nodes with various features.

For information how to start using ESP32 and ESP-BLE-MESH, please see the Section *Getting Started with ESP-BLE-MESH*. If you are interested in information on ESP-BLE-MESH architecture, including some details of software implementation, please see Section *ESP-BLE-MESH Architecture*.

Application Examples and Demos

Please refer to Sections *ESP-BLE-MESH Examples* and *ESP-BLE-MESH Demo Videos*.

API Reference

ESP-BLE-MESH APIs are divided into the following parts:

- *ESP-BLE-MESH Definitions*
- *ESP-BLE-MESH Core API Reference*
- *ESP-BLE-MESH Models API Reference*

ESP-BLE-MESH Definitions

This section contains only one header file, which lists the following items of ESP-BLE-MESH.

- ID of all the models and related message opcodes
- Structs of model, element and Composition Data
- Structs of used by ESP-BLE-MESH Node/Provisioner for provisioning
- Structs used to transmit/receive messages
- Event types and related event parameters

Header File

- components/bt/esp_ble_mesh/api/esp_ble_mesh_defs.h

Unions

union **esp_ble_mesh_prov_cb_param_t**

#include <esp_ble_mesh_defs.h> BLE Mesh Node/Provisioner callback parameters union.

Public Members

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_prov_register_comp_param* **prov_register_comp**

Event parameter of ESP_BLE_MESH_PROV_REGISTER_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_set_unprov_dev_name_comp_param* **node_set_unprov_dev_name_comp**

Event parameter of ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_prov_enable_comp_param* **node_prov_enable_comp**

Event parameter of ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_prov_disable_comp_param* **node_prov_disable_comp**

Event parameter of ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_link_open_evt_param* **node_prov_link_open**

Event parameter of ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_link_close_evt_param* **node_prov_link_close**

Event parameter of ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_output_num_evt_param* **node_prov_output_num**

Event parameter of ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_output_str_evt_param* **node_prov_output_str**

Event parameter of ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_input_evt_param* **node_prov_input**

Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provision_complete_evt_param* **node_prov_complete**

Event parameter of ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provision_reset_param* **node_prov_reset**

Event parameter of ESP_BLE_MESH_NODE_PROV_RESET_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_set_oob_pub_key_comp_param* **node_prov_set_oob_pub_key_comp**

Event parameter of ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_input_number_comp_param* **node_prov_input_num_comp**

Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_NUM_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_input_string_comp_param* **node_prov_input_str_comp**

Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_STR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_identity_enable_comp_param* **node_proxy_identity_enable_comp**

Event parameter of ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_gatt_enable_comp_param*
node_proxy_gatt_enable_comp

Event parameter of ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_gatt_disable_comp_param*
node_proxy_gatt_disable_comp

Event parameter of ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_node_add_local_net_key_comp_param*
node_add_net_key_comp

Event parameter of ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_node_add_local_app_key_comp_param*
node_add_app_key_comp

Event parameter of ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_node_bind_local_mod_app_comp_param*
node_bind_app_key_to_model_comp

Event parameter of ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_recv_unprov_adv_pkt_param*
provisioner_recv_unprov_adv_pkt

Event parameter of ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_enable_comp_param*
provisioner_prov_enable_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_disable_comp_param*
provisioner_prov_disable_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_link_open_evt_param*
provisioner_prov_link_open

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_read_oob_pub_key_evt_param*
provisioner_prov_read_oob_pub_key

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_input_evt_param*
provisioner_prov_input

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_output_evt_param*
provisioner_prov_output

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_link_close_evt_param*
provisioner_prov_link_close

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_comp_param*
provisioner_prov_complete

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_add_unprov_dev_comp_param*
provisioner_add_unprov_dev_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_dev_with_addr_comp_param*
provisioner_prov_dev_with_addr_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_delete_dev_comp_param*
provisioner_delete_dev_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_set_dev_uuid_match_comp_param*
provisioner_set_dev_uuid_match_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_set_prov_data_info_comp_param*
provisioner_set_prov_data_info_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_set_static_oob_val_comp_param*
provisioner_set_static_oob_val_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_set_primary_elem_addr_comp_param*
provisioner_set_primary_elem_addr_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_read_oob_pub_key_comp_param*
provisioner_prov_read_oob_pub_key_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_input_num_comp_param*
provisioner_prov_input_num_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_input_str_comp_param*
provisioner_prov_input_str_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_set_node_name_comp_param*
provisioner_set_node_name_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_add_local_app_key_comp_param*
provisioner_add_app_key_comp

Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_update_local_app_key_comp_param* **provisioner_update_app_key_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_bind_local_mod_app_comp_param* **provisioner_bind_app_key_to_model_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_add_local_net_key_comp_param* **provisioner_add_net_key_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_update_local_net_key_comp_param* **provisioner_update_net_key_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_store_node_comp_data_comp_param* **provisioner_store_node_comp_data_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_delete_node_with_uuid_comp_param* **provisioner_delete_node_with_uuid_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_delete_node_with_addr_comp_param* **provisioner_delete_node_with_addr_comp**

Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT

int **err_code**

Indicate the result of enabling/disabling to receive heartbeat messages by the Provisioner

Indicate the result of setting the heartbeat filter type by the Provisioner

Indicate the result of setting the heartbeat filter address by the Provisioner

Indicate the result of directly erasing settings by the Provisioner

Indicate the result of opening settings with index by the Provisioner

Indicate the result of opening settings with user id by the Provisioner

Indicate the result of closing settings with index by the Provisioner

Indicate the result of closing settings with user id by the Provisioner

Indicate the result of deleting settings with index by the Provisioner

Indicate the result of deleting settings with user id by the Provisioner

bool **enable**

Indicate enabling or disabling receiving heartbeat messages

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_enable_heartbeat_recv_comp**

ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT

uint8_t **type**

Type of the filter used for receiving heartbeat messages

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_set_heartbeat_filter_type_comp**

ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT

uint8_t **op**

Operation (add, remove, clean)

uint16_t **hb_src**

Heartbeat source address

uint16_t **hb_dst**

Heartbeat destination address

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_set_heartbeat_filter_info_comp**

ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT

uint8_t **init_ttl**

Heartbeat InitTTL

uint8_t **rx_ttl**

Heartbeat RxTTL

uint8_t **hops**

Heartbeat hops (InitTTL - RxTTL + 1)

uint16_t **feature**

Bit field of currently active features of the node

int8_t **rssi**

RSSI of the heartbeat message

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_recv_heartbeat**

ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_direct_erase_settings_comp**

ESP_BLE_MESH_PROVISIONER_DIRECT_ERASE_SETTINGS_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_DIRECT_ERASE_SETTINGS_COMP_EVT

uint8_t **index**

Index of Provisioner settings

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_open_settings_with_index_comp**

ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT.

Event parameter of ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT

char **uid**[ESP_BLE_MESH_SETTINGS_UID_SIZE + 1]

Provisioner settings user id

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_open_settings_with_uid_comp**

ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_close_settings_with_index_comp**

ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT.

Event parameter of ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_close_settings_with_uid_comp**

ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_delete_settings_with_index_comp**

ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT.

Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::[anonymous] **provisioner_delete_settings_with_uid_comp**

ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_set_fast_prov_info_comp_param* **set_fast_prov_info_comp**

Event parameter of ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_set_fast_prov_action_comp_param* **set_fast_prov_action_comp**

Event parameter of ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_heartbeat_msg_recv_param* **heartbeat_msg_recv**

Event parameter of ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_lpn_enable_comp_param* **lpn_enable_comp**

Event parameter of ESP_BLE_MESH_LPN_ENABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_lpn_disable_comp_param* **lpn_disable_comp**

Event parameter of ESP_BLE_MESH_LPN_DISABLE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_lpn_poll_comp_param* **lpn_poll_comp**

Event parameter of ESP_BLE_MESH_LPN_POLL_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_lpn_friendship_establish_param* **lpn_friendship_establish**

Event parameter of ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_lpn_friendship_terminate_param* **lpn_friendship_terminate**

Event parameter of ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_friend_friendship_establish_param* **frnd_friendship_establish**

Event parameter of ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_friend_friendship_terminate_param* **frnd_friendship_terminate**

Event parameter of ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_recv_adv_pkt_param* **proxy_client_recv_adv_pkt**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_connected_param* **proxy_client_connected**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_disconnected_param* **proxy_client_disconnected**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_recv_filter_status_param* **proxy_client_recv_filter_status**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_connect_comp_param* **proxy_client_connect_comp**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_disconnect_comp_param* **proxy_client_disconnect_comp**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_set_filter_type_comp_param* **proxy_client_set_filter_type_comp**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_add_filter_addr_comp_param* **proxy_client_add_filter_addr_comp**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_client_remove_filter_addr_comp_param* **proxy_client_remove_filter_addr_comp**

Event parameter of ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_server_connected_param* **proxy_server_connected**

Event parameter of ESP_BLE_MESH_PROXY_SERVER_CONNECTED_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_proxy_server_disconnected_param* **proxy_server_disconnected**

Event parameter of ESP_BLE_MESH_PROXY_SERVER_DISCONNECTED_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_model_sub_group_addr_comp_param* **model_sub_group_addr_comp**

Event parameters of ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_model_unsub_group_addr_comp_param* **model_unsub_group_addr_comp**

Event parameters of ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_deinit_mesh_comp_param* **deinit_mesh_comp**

Event parameter of ESP_BLE_MESH_DEINIT_MESH_COMP_EVT

struct **ble_mesh_deinit_mesh_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_DEINIT_MESH_COMP_EVT.

Public Members

int **err_code**

Indicate the result of BLE Mesh deinitialization

struct **ble_mesh_friend_friendship_establish_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT.

Public Members

uint16_t **lpn_addr**

Low Power Node unicast address

struct **ble_mesh_friend_friendship_terminate_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT.

Public Types

enum **[anonymous]**

This enum value is the reason of friendship termination on the friend node side

Values:

enumerator **ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_ESTABLISH_FAIL**

Friend Offer has been sent, but Friend Offer is not received within 1 second, friendship fails to be established

enumerator **ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_POLL_TIMEOUT**

Friendship is established, PollTimeout timer expires and no Friend Poll/Sub Add/Sub Remove is received

enumerator **ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_RECV_FRND_REQ**

Receive Friend Request from existing Low Power Node

enumerator **ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_RECV_FRND_CLEAR**

Receive Friend Clear from other friend node

enumerator **ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_DISABLE**

Friend feature disabled or corresponding NetKey is deleted

Public Members

uint16_t **lpn_addr**

Low Power Node unicast address

enum *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_friend_friendship_terminate_param*::*[anonymous]* **reason**

This enum value is the reason of friendship termination on the friend node side Friendship terminated reason

struct **ble_mesh_heartbeat_msg_recv_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT.

Public Members

uint8_t **hops**

Heartbeat hops (InitTTL - RxTTL + 1)

uint16_t **feature**

Bit field of currently active features of the node

struct **ble_mesh_input_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_EVT.

Public Members

esp_ble_mesh_input_action_t **action**

Action of Input OOB Authentication

uint8_t **size**

Size of Input OOB Authentication

struct **ble_mesh_input_number_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_NUM_COMP_EVT.

Public Members

int **err_code**

Indicate the result of inputting number

struct **ble_mesh_input_string_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_STR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of inputting string

struct **ble_mesh_link_close_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**

Type of the bearer used when device link is closed

struct **ble_mesh_link_open_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**

Type of the bearer used when device link is open

struct **ble_mesh_lpn_disable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_DISABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of disabling LPN functionality

struct **ble_mesh_lpn_enable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_ENABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of enabling LPN functionality

struct **ble_mesh_lpn_friendship_establish_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT.

Public Members

uint16_t **friend_addr**

Friend Node unicast address

struct **ble_mesh_lpn_friendship_terminate_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT.

Public Members

uint16_t **friend_addr**

Friend Node unicast address

struct **ble_mesh_lpn_poll_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_POLL_COMP_EVT.

Public Members

int **err_code**

Indicate the result of sending Friend Poll

struct **ble_mesh_model_sub_group_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of local model subscribing group address

uint16_t **element_addr**

Element address

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

uint16_t **group_addr**

Group Address

struct **ble_mesh_model_unsub_group_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of local model unsubscribing group address

uint16_t **element_addr**

Element address

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

uint16_t **group_addr**

Group Address

struct **ble_mesh_node_add_local_app_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local AppKey by the node

uint16_t **net_idx**

NetKey Index

uint16_t **app_idx**

AppKey Index

struct **ble_mesh_node_add_local_net_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local NetKey by the node

uint16_t **net_idx**

NetKey Index

struct **ble_mesh_node_bind_local_mod_app_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT.

Public Members

int **err_code**

Indicate the result of binding AppKey with model by the node

uint16_t **element_addr**

Element address

uint16_t **app_idx**

AppKey Index

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **ble_mesh_output_num_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT.

Public Members

esp_ble_mesh_output_action_t **action**

Action of Output OOB Authentication

uint32_t **number**

Number of Output OOB Authentication

struct **ble_mesh_output_str_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT.

Public Members

char **string**[8]

String of Output OOB Authentication

struct **ble_mesh_prov_disable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of disabling BLE Mesh device

struct **ble_mesh_prov_enable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of enabling BLE Mesh device

struct **ble_mesh_prov_register_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROV_REGISTER_COMP_EVT.

Public Members

int **err_code**

Indicate the result of BLE Mesh initialization

struct **ble_mesh_provision_complete_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT.

Public Members

uint16_t **net_idx**

NetKey Index

uint8_t **net_key**[16]

NetKey

uint16_t **addr**

Primary address

uint8_t **flags**

Flags

uint32_t **iv_index**

IV Index

struct **ble_mesh_provision_reset_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_RESET_EVT.

struct **ble_mesh_provisioner_add_local_app_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local AppKey by the Provisioner

uint16_t **net_idx**

NetKey Index

uint16_t **app_idx**

AppKey Index

struct **ble_mesh_provisioner_add_local_net_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local NetKey by the Provisioner

uint16_t **net_idx**

NetKey Index

struct **ble_mesh_provisioner_add_unprov_dev_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding device into queue by the Provisioner

struct **ble_mesh_provisioner_bind_local_mod_app_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT.

Public Members

int **err_code**

Indicate the result of binding AppKey with model by the Provisioner

uint16_t **element_addr**

Element address

uint16_t **app_idx**

AppKey Index

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **ble_mesh_provisioner_delete_dev_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT.

Public Members

int **err_code**

Indicate the result of deleting device by the Provisioner

struct **ble_mesh_provisioner_delete_node_with_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of deleting node with unicast address by the Provisioner

uint16_t **unicast_addr**

Node unicast address

struct **ble_mesh_provisioner_delete_node_with_uuid_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT.

Public Members

int **err_code**

Indicate the result of deleting node with uuid by the Provisioner

uint8_t **uuid**[16]

Node device uuid

struct **ble_mesh_provisioner_link_close_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**

Type of the bearer used when Provisioner link is closed

uint8_t **reason**

Reason of the closed provisioning link

struct **ble_mesh_provisioner_link_open_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**

Type of the bearer used when Provisioner link is opened

struct **ble_mesh_provisioner_prov_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Public Members

uint16_t **node_idx**

Index of the provisioned device

esp_ble_mesh_octet16_t **device_uuid**

Device UUID of the provisioned device

uint16_t **unicast_addr**

Primary address of the provisioned device

uint8_t **element_num**

Element count of the provisioned device

uint16_t **netkey_idx**

NetKey Index of the provisioned device

struct **ble_mesh_provisioner_prov_dev_with_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Provisioner starting to provision a device

struct **ble_mesh_provisioner_prov_disable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of disabling BLE Mesh Provisioner

struct **ble_mesh_provisioner_prov_enable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of enabling BLE Mesh Provisioner

struct **ble_mesh_provisioner_prov_input_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT.

Public Members

esp_ble_mesh_oob_method_t **method**

Method of device Output OOB Authentication

esp_ble_mesh_output_action_t **action**

Action of device Output OOB Authentication

uint8_t **size**

Size of device Output OOB Authentication

uint8_t **link_idx**

Index of the provisioning link

struct **ble_mesh_provisioner_prov_input_num_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT.

Public Members

int **err_code**

Indicate the result of inputting number by the Provisioner

struct **ble_mesh_provisioner_prov_input_str_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT.

Public Members

int **err_code**

Indicate the result of inputting string by the Provisioner

struct **ble_mesh_provisioner_prov_output_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT.

Public Members

esp_ble_mesh_oob_method_t **method**

Method of device Input OOB Authentication

esp_ble_mesh_input_action_t **action**

Action of device Input OOB Authentication

uint8_t **size**

Size of device Input OOB Authentication

uint8_t **link_idx**

Index of the provisioning link

char **string**[8]

String output by the Provisioner

uint32_t **number**

Number output by the Provisioner

union *esp_ble_mesh_prov_cb_param_t*::*ble_mesh_provisioner_prov_output_evt_param*::[anonymous] **[anonymous]**

struct **ble_mesh_provisioner_prov_read_oob_pub_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting OOB Public Key by the Provisioner

struct **ble_mesh_provisioner_prov_read_oob_pub_key_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT.

Public Members

uint8_t **link_idx**

Index of the provisioning link

struct **ble_mesh_provisioner_recv_unprov_adv_pkt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT.

Public Members

uint8_t **dev_uuid**[16]

Device UUID of the unprovisioned device

esp_ble_mesh_bd_addr_t **addr**

Device address of the unprovisioned device

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **oob_info**

OOB Info of the unprovisioned device

uint8_t **adv_type**

Avertising type of the unprovisioned device

esp_ble_mesh_prov_bearer_t **bearer**

Bearer of the unprovisioned device

int8_t **rssi**

RSSI of the received advertising packet

struct **ble_mesh_provisioner_set_dev_uuid_match_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting Device UUID match value by the Provisioner

struct **ble_mesh_provisioner_set_node_name_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting provisioned device name by the Provisioner

uint16_t **node_index**

Index of the provisioned device

struct **ble_mesh_provisioner_set_primary_elem_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting unicast address of primary element by the Provisioner

struct **ble_mesh_provisioner_set_prov_data_info_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting provisioning info by the Provisioner

struct **ble_mesh_provisioner_set_static_oob_val_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting static oob value by the Provisioner

struct **ble_mesh_provisioner_store_node_comp_data_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT.

Public Members

int **err_code**

Indicate the result of storing node composition data by the Provisioner

uint16_t **addr**

Node element address

struct **ble_mesh_provisioner_update_local_app_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of updating local AppKey by the Provisioner

uint16_t **net_idx**

NetKey Index

uint16_t **app_idx**

AppKey Index

struct **ble_mesh_provisioner_update_local_net_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of updating local NetKey by the Provisioner

uint16_t **net_idx**

NetKey Index

struct **ble_mesh_proxy_client_add_filter_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client add filter address

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct **ble_mesh_proxy_client_connect_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client connect

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **net_idx**

Corresponding NetKey Index

struct **ble_mesh_proxy_client_connected_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct **ble_mesh_proxy_client_disconnect_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client disconnect

uint8_t **conn_handle**

Proxy connection handle

struct **ble_mesh_proxy_client_disconnected_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

uint8_t **reason**

Proxy disconnect reason

struct **ble_mesh_proxy_client_recv_adv_pkt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **net_idx**

Network ID related NetKey Index

uint8_t **net_id**[8]

Network ID contained in the advertising packet

int8_t **rssi**

RSSI of the received advertising packet

struct **ble_mesh_proxy_client_recv_filter_status_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT.

Public Members

uint8_t **conn_handle**

Proxy connection handle

uint16_t **server_addr**

Proxy Server primary element address

uint16_t **net_idx**

Corresponding NetKey Index

uint8_t **filter_type**

Proxy Server filter type(whitelist or blacklist)

uint16_t **list_size**

Number of addresses in the Proxy Server filter list

struct **ble_mesh_proxy_client_remove_filter_addr_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client remove filter address

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct **ble_mesh_proxy_client_set_filter_type_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client set filter type

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct **ble_mesh_proxy_gatt_disable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of disabling Mesh Proxy Service

struct **ble_mesh_proxy_gatt_enable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of enabling Mesh Proxy Service

struct **ble_mesh_proxy_identity_enable_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of enabling Mesh Proxy advertising

struct **ble_mesh_proxy_server_connected_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_SERVER_CONNECTED_EVT.

Public Members

uint8_t **conn_handle**

Proxy connection handle

struct **ble_mesh_proxy_server_disconnected_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_SERVER_DISCONNECTED_EVT.

Public Members

uint8_t **conn_handle**

Proxy connection handle

uint8_t **reason**

Proxy disconnect reason

struct **ble_mesh_set_fast_prov_action_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT.

Public Members

uint8_t **status_action**

Indicate the result of setting action of fast provisioning

struct **ble_mesh_set_fast_prov_info_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT.

Public Members

uint8_t **status_unicast**

Indicate the result of setting unicast address range of fast provisioning

uint8_t **status_net_idx**

Indicate the result of setting NetKey Index of fast provisioning

uint8_t **status_match**

Indicate the result of setting matching Device UUID of fast provisioning

struct **ble_mesh_set_oob_pub_key_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting OOB Public Key

struct **ble_mesh_set_unprov_dev_name_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting BLE Mesh device name

union **esp_ble_mesh_server_state_value_t**

#include <esp_ble_mesh_defs.h> Server model state value union.

Public Members

uint8_t **onoff**

The value of the Generic OnOff state

The value of the Light LC Light OnOff state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **gen_onoff**

The Generic OnOff state

int16_t **level**

The value of the Generic Level state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **gen_level**

The Generic Level state

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **gen_onpowerup**

The Generic OnPowerUp state

uint16_t **power**

The value of the Generic Power Actual state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **gen_power_actual**

The Generic Power Actual state

uint16_t **lightness**

The value of the Light Lightness Actual state

The value of the Light Lightness Linear state

The value of the Light CTL Lightness state

The value of the Light HSL Lightness state

The value of the Light xyL Lightness state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_lightness_actual**

The Light Lightness Actual state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_lightness_linear**

The Light Lightness Linear state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_ctl_lightness**

The Light CTL Lightness state

uint16_t **temperature**

The value of the Light CTL Temperature state

int16_t **delta_uv**

The value of the Light CTL Delta UV state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_ctl_temp_delta_uv**

The Light CTL Temperature & Delta UV states

uint16_t **hue**

The value of the Light HSL Hue state

uint16_t **saturation**

The value of the Light HSL Saturation state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_hsl**

The Light HSL composite state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_hsl_lightness**

The Light HSL Lightness state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_hsl_hue**

The Light HSL Hue state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_hsl_saturation**

The Light HSL Saturation state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_xyl_lightness**

The Light xyL Lightness state

struct *esp_ble_mesh_server_state_value_t*::[anonymous] **light_lc_light_onoff**

The Light LC Light OnOff state

union **esp_ble_mesh_model_cb_param_t**

#include <esp_ble_mesh_defs.h> BLE Mesh model callback parameters union.

Public Members

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_model_operation_evt_param* **model_operation**

Event parameter of ESP_BLE_MESH_MODEL_OPERATION_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_model_send_comp_param* **model_send_comp**

Event parameter of ESP_BLE_MESH_MODEL_SEND_COMP_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_model_publish_comp_param* **model_publish_comp**

Event parameter of ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_mod_recv_publish_msg_param*
client_recv_publish_msg

Event parameter of ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_client_model_send_timeout_param*
client_send_timeout

Event parameter of ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_model_publish_update_evt_param*
model_publish_update

Event parameter of ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT

struct *esp_ble_mesh_model_cb_param_t*::*ble_mesh_server_model_update_state_comp_param*
server_model_update_state

Event parameter of ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT

struct **ble_mesh_client_model_send_timeout_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT.

Public Members

uint32_t **opcode**

Opcode of the previously sent message

esp_ble_mesh_model_t ***model**

Pointer to the model which sends the previous message

esp_ble_mesh_msg_ctx_t ***ctx**

Pointer to the context of the previous message

struct **ble_mesh_mod_recv_publish_msg_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT.

Public Members

uint32_t **opcode**

Opcode of the unsolicited received message

esp_ble_mesh_model_t ***model**

Pointer to the model which receives the message

esp_ble_mesh_msg_ctx_t ***ctx**

Pointer to the context of the message

uint16_t **length**

Length of the received message

uint8_t ***msg**

Value of the received message

struct **ble_mesh_model_operation_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_OPERATION_EVT.

Public Members

uint32_t **opcode**

Opcode of the received message

esp_ble_mesh_model_t ***model**

Pointer to the model which receives the message

esp_ble_mesh_msg_ctx_t ***ctx**

Pointer to the context of the received message

uint16_t **length**

Length of the received message

uint8_t ***msg**

Value of the received message

struct **ble_mesh_model_publish_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT.

Public Members

int **err_code**

Indicate the result of publishing a message

esp_ble_mesh_model_t ***model**

Pointer to the model which publishes the message

struct **ble_mesh_model_publish_update_evt_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT.

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the model which is going to update its publish message

struct **ble_mesh_model_send_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_SEND_COMP_EVT.

Public Members

int **err_code**

Indicate the result of sending a message

uint32_t **opcode**

Opcode of the message

esp_ble_mesh_model_t ***model**

Pointer to the model which sends the message

esp_ble_mesh_msg_ctx_t ***ctx**

Context of the message

struct **ble_mesh_server_model_update_state_comp_param**

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of updating server model state

esp_ble_mesh_model_t ***model**

Pointer to the server model which state value is updated

esp_ble_mesh_server_state_type_t **type**

Type of the updated server state

Structures

struct **esp_ble_mesh_deinit_param_t**

BLE Mesh deinit parameters

Public Members

bool **erase_flash**

Indicate if erasing flash when deinit mesh stack

struct **esp_ble_mesh_elem_t**

Abstraction that describes a BLE Mesh Element. This structure is associated with struct bt_mesh_elem in mesh_access.h

Public Members

uint16_t **element_addr**

Element Address, assigned during provisioning.

const uint16_t **location**

Location Descriptor (GATT Bluetooth Namespace Descriptors)

const uint8_t **sig_model_count**

SIG Model count

const uint8_t **vnd_model_count**

Vendor Model count

esp_ble_mesh_model_t ***sig_models**

SIG Models

esp_ble_mesh_model_t ***vnd_models**

Vendor Models

struct **esp_ble_mesh_model_pub_t**

Abstraction that describes a model publication context. This structure is associated with struct bt_mesh_model_pub in mesh_access.h

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the model to which the context belongs. Initialized by the stack.

uint16_t **publish_addr**

Publish Address.

uint16_t **app_idx**

Publish AppKey Index.

uint16_t **cred**

Friendship Credentials Flag.

uint16_t **send_rel**

Force reliable sending (segment acks)

uint8_t **ttl**

Publish Time to Live.

uint8_t **retransmit**

Retransmit Count & Interval Steps.

uint8_t **period**

Publish Period.

uint8_t **period_div**

Divisor for the Period.

uint8_t **fast_period**

Use FastPeriodDivisor

uint8_t **count**

Retransmissions left.

uint32_t **period_start**

Start of the current period.

struct net_buf_simple ***msg**

Publication buffer, containing the publication message.

This will get correctly created when the publication context has been defined using the ESP_BLE_MESH_MODEL_PUB_DEFINE macro.

ESP_BLE_MESH_MODEL_PUB_DEFINE(name, size);

esp_ble_mesh_cb_t **update**

Callback used to update publish message. Initialized by the stack.

struct k_delayed_work **timer**

Publish Period Timer. Initialized by the stack.

uint8_t **dev_role**

Role of the device that is going to publish messages

struct **esp_ble_mesh_model_op_t**

Abstraction that describes a model operation context. This structure is associated with struct bt_mesh_model_op in mesh_access.h

Public Members

const uint32_t **opcode**

Message opcode

const size_t **min_len**

Message minimum length

esp_ble_mesh_cb_t **param_cb**

Callback used to handle message. Initialized by the stack.

struct **esp_ble_mesh_model_cbs_t**

Abstraction that describes a model callback structure. This structure is associated with struct bt_mesh_model_cb in mesh_access.h.

Public Members

esp_ble_mesh_cb_t **init_cb**

Callback used during model initialization. Initialized by the stack.

struct **esp_ble_mesh_model**

Abstraction that describes a Mesh Model instance. This structure is associated with struct bt_mesh_model in mesh_access.h

Public Members

const uint16_t **model_id**

16-bit model identifier

uint16_t **company_id**

16-bit company identifier

uint16_t **model_id**

16-bit model identifier

struct *esp_ble_mesh_model*::[anonymous]::[anonymous] **vnd**

Structure encapsulating a model ID with a company ID

union *esp_ble_mesh_model*::[anonymous] **[anonymous]**

Model ID

uint8_t **element_idx**

Internal information, mainly for persistent storage Belongs to Nth element

uint8_t **model_idx**

Is the Nth model in the element

uint16_t **flags**

Information about what has changed

esp_ble_mesh_elem_t ***element**

The Element to which this Model belongs

esp_ble_mesh_model_pub_t *const **pub**

Model Publication

uint16_t **keys**[CONFIG_BLE_MESH_MODEL_KEY_COUNT]

AppKey List

uint16_t **groups**[CONFIG_BLE_MESH_MODEL_GROUP_COUNT]

Subscription List (group or virtual addresses)

esp_ble_mesh_model_op_t ***op**

Model operation context

esp_ble_mesh_model_cbs_t ***cb**

Model callback structure

void ***user_data**

Model-specific user data

struct **esp_ble_mesh_msg_ctx_t**

Message sending context. This structure is associated with struct bt_mesh_msg_ctx in mesh_access.h

Public Members

uint16_t **net_idx**

NetKey Index of the subnet through which to send the message.

uint16_t **app_idx**

AppKey Index for message encryption.

uint16_t **addr**

Remote address.

uint16_t **recv_dst**

Destination address of a received message. Not used for sending.

int8_t **recv_rssi**

RSSI of received packet. Not used for sending.

uint8_t **recv_ttl**

Received TTL value. Not used for sending.

uint8_t **send_rel**

Force sending reliably by using segment acknowledgement

uint8_t **send_ttl**

TTL, or ESP_BLE_MESH_TTL_DEFAULT for default TTL.

uint32_t **recv_op**

Opcode of a received message. Not used for sending message.

esp_ble_mesh_model_t ***model**

Model corresponding to the message, no need to be initialized before sending message

bool **srv_send**

Indicate if the message is sent by a node server model, no need to be initialized before sending message

struct **esp_ble_mesh_prov_t**

Provisioning properties & capabilities. This structure is associated with struct bt_mesh_prov in mesh_access.h

struct **esp_ble_mesh_comp_t**

Node Composition data context. This structure is associated with struct bt_mesh_comp in mesh_access.h

Public Members

uint16_t **cid**

16-bit SIG-assigned company identifier

uint16_t **pid**

16-bit vendor-assigned product identifier

uint16_t **vid**

16-bit vendor-assigned product version identifier

size_t **element_count**

Element count

esp_ble_mesh_elem_t ***elements**

A sequence of elements

struct **esp_ble_mesh_unprov_dev_add_t**

Information of the device which is going to be added for provisioning.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **uuid**[16]

Device UUID

uint16_t **oob_info**

Device OOB Info ADD_DEV_START_PROV_NOW_FLAG shall not be set if the bearer has both PB-ADV and PB-GATT enabled

esp_ble_mesh_prov_bearer_t **bearer**

Provisioning Bearer

struct **esp_ble_mesh_device_delete_t**

Information of the device which is going to be deleted.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **uuid**[16]

Device UUID

uint8_t **flag**

BIT0: device address; BIT1: device UUID

struct **esp_ble_mesh_prov_data_info_t**

Information of the provisioner which is going to be updated.

Public Members

uint16_t **net_idx**

NetKey Index

uint8_t **flags**

Flags

uint32_t **iv_index**

IV Index

uint8_t **flag**

BIT0: net_idx; BIT1: flags; BIT2: iv_index

struct **esp_ble_mesh_node_t**

Information of the provisioned node

Public Members

esp_ble_mesh_bd_addr_t **addr**

Node device address

esp_ble_mesh_addr_type_t **addr_type**

Node device address type

uint8_t **dev_uuid**[16]

Device UUID

uint16_t **oob_info**

Node OOB information

uint16_t **unicast_addr**

Node unicast address

uint8_t **element_num**

Node element number

uint16_t **net_idx**

Node NetKey Index

uint8_t **flags**

Node key refresh flag and iv update flag

uint32_t **iv_index**

Node IV Index

uint8_t **dev_key**[16]

Node device key

char **name**[ESP_BLE_MESH_NODE_NAME_MAX_LEN + 1]

Node name

uint16_t **comp_length**

Length of Composition Data

uint8_t ***comp_data**

Value of Composition Data

struct **esp_ble_mesh_fast_prov_info_t**

Context of fast provisioning which need to be set.

Public Members

uint16_t **unicast_min**

Minimum unicast address used for fast provisioning

uint16_t **unicast_max**

Maximum unicast address used for fast provisioning

uint16_t **net_idx**

Netkey index used for fast provisioning

uint8_t **flags**

Flags used for fast provisioning

uint32_t **iv_index**

IV Index used for fast provisioning

uint8_t **offset**

Offset of the UUID to be compared

uint8_t **match_len**

Length of the UUID to be compared

uint8_t **match_val**[16]

Value of UUID to be compared

struct **esp_ble_mesh_heartbeat_filter_info_t**

Context of Provisioner heartbeat filter information to be set

Public Members

uint16_t **hb_src**

Heartbeat source address (unicast address)

uint16_t **hb_dst**

Heartbeat destination address (unicast address or group address)

struct **esp_ble_mesh_client_op_pair_t**

BLE Mesh client models related definitions.

Client model Get/Set message opcode and corresponding Status message opcode

Public Members

uint32_t **cli_op**

The client message opcode

uint32_t **status_op**

The server status opcode corresponding to the client message opcode

struct **esp_ble_mesh_client_t**

Client Model user data context.

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the client model. Initialized by the stack.

int **op_pair_size**

Size of the op_pair

const *esp_ble_mesh_client_op_pair_t* ***op_pair**

Table containing get/set message opcode and corresponding status message opcode

uint32_t **publish_status**

Callback used to handle the received unsolicited message. Initialized by the stack.

void ***internal_data**

Pointer to the internal data of client model

uint8_t **msg_role**

Role of the device (Node/Provisioner) that is going to send messages

struct **esp_ble_mesh_client_common_param_t**

Common parameters of the messages sent by Client Model.

Public Members

esp_ble_mesh_opcode_t **opcode**

Message opcode

esp_ble_mesh_model_t ***model**

Pointer to the client model structure

esp_ble_mesh_msg_ctx_t **ctx**

The context used to send message

int32_t **msg_timeout**

Timeout value (ms) to get response to the sent message Note: if using default timeout value in menuconfig, make sure to set this value to 0

uint8_t **msg_role**

Role of the device - Node/Provisioner

struct **esp_ble_mesh_state_transition_t**

Parameters of the server model state transition

Public Functions

BLE_MESH_ATOMIC_DEFINE(flag, *ESP_BLE_MESH_SERVER_FLAG_MAX*)

Flag used to indicate if the transition timer has been started internally.

If the model which contains *esp_ble_mesh_state_transition_t* sets "set_auto_rsp" to ESP_BLE_MESH_SERVER_RSP_BY_APP, the handler of the timer shall be initialized by the users.

And users can use this flag to indicate whether the timer is started or not.

Public Members

bool **just_started**

Indicate if the state transition has just started

uint8_t **trans_time**

State transition time

uint8_t **remain_time**

Remaining time of state transition

uint8_t **delay**

Delay before starting state transition

uint32_t **quo_tt**

Duration of each divided transition step

uint32_t **counter**

Number of steps which the transition duration is divided

uint32_t **total_duration**

State transition total duration

int64_t **start_timestamp**

Time when the state transition is started

struct k_delayed_work **timer**

Timer used for state transition

struct **esp_ble_mesh_last_msg_info_t**

Parameters of the server model received last same set message.

Public Members

uint8_t **tid**

Transaction number of the last message

uint16_t **src**

Source address of the last message

uint16_t **dst**

Destination address of the last message

int64_t **timestamp**

Time when the last message is received

struct **esp_ble_mesh_server_rsp_ctrl_t**

Parameters of the Server Model response control

Public Members

uint8_t **get_auto_rsp**

BLE Mesh Server Response Option.

i. If get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, then the response of Client Get messages need to be replied by the application;
ii. If get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, then the response of Client Get messages will be replied by the server models;

iii. If set_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, then the response of Client Set messages need to be replied by the application;
iv. If set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, then the response of Client Set messages will be replied by the server models;
v. If status_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, then the response of Server Status messages need to be replied by the application;
vi. If status_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, then the response of Server Status messages will be replied by the server models; Response control for Client Get messages

uint8_t **set_auto_rsp**

Response control for Client Set messages

uint8_t **status_auto_rsp**

Response control for Server Status messages

Macros

ESP_BLE_MESH_SDU_MAX_LEN

< The maximum length of a BLE Mesh message, including Opcode, Payload and TransMIC Length of a short Mesh MIC.

ESP_BLE_MESH_MIC_SHORT

Length of a long Mesh MIC.

ESP_BLE_MESH_MIC_LONG

The maximum length of a BLE Mesh provisioned node name

ESP_BLE_MESH_NODE_NAME_MAX_LEN

The maximum length of a BLE Mesh unprovisioned device name

ESP_BLE_MESH_DEVICE_NAME_MAX_LEN

The maximum length of settings user id

ESP_BLE_MESH_SETTINGS_UID_SIZE

Invalid settings index

ESP_BLE_MESH_INVALID_SETTINGS_IDX

Define the BLE Mesh octet 16 bytes size

ESP_BLE_MESH_OCTET16_LEN

ESP_BLE_MESH_OCTET8_LEN

ESP_BLE_MESH_CID_NVAL

Special TTL value to request using configured default TTL

ESP_BLE_MESH_TTL_DEFAULT

Maximum allowed TTL value

ESP_BLE_MESH_TTL_MAX

ESP_BLE_MESH_ADDR_UNASSIGNED

ESP_BLE_MESH_ADDR_ALL_NODES

ESP_BLE_MESH_ADDR_PROXIES

ESP_BLE_MESH_ADDR_FRIENDS

ESP_BLE_MESH_ADDR_RELAYS

ESP_BLE_MESH_KEY_UNUSED

ESP_BLE_MESH_KEY_DEV

ESP_BLE_MESH_KEY_PRIMARY

ESP_BLE_MESH_KEY_ANY

Internal macros used to initialize array members

ESP_BLE_MESH_KEY_UNUSED_ELT_(IDX, _)

ESP_BLE_MESH_ADDR_UNASSIGNED_ELT_(IDX, _)

ESP_BLE_MESH_MODEL_KEYS_UNUSED

ESP_BLE_MESH_MODEL_GROUPS_UNASSIGNED

Primary Network Key index

ESP_BLE_MESH_NET_PRIMARY

Relay state value

ESP_BLE_MESH_RELAY_DISABLED

ESP_BLE_MESH_RELAY_ENABLED

ESP_BLE_MESH_RELAY_NOT_SUPPORTED

Beacon state value

ESP_BLE_MESH_BEACON_DISABLED

ESP_BLE_MESH_BEACON_ENABLED

GATT Proxy state value

ESP_BLE_MESH_GATT_PROXY_DISABLED

ESP_BLE_MESH_GATT_PROXY_ENABLED

ESP_BLE_MESH_GATT_PROXY_NOT_SUPPORTED

Friend state value

ESP_BLE_MESH_FRIEND_DISABLED

ESP_BLE_MESH_FRIEND_ENABLED

ESP_BLE_MESH_FRIEND_NOT_SUPPORTED

Node identity state value

ESP_BLE_MESH_NODE_IDENTITY_STOPPED

ESP_BLE_MESH_NODE_IDENTITY_RUNNING

ESP_BLE_MESH_NODE_IDENTITY_NOT_SUPPORTED

Supported features

ESP_BLE_MESH_FEATURE_RELAY

ESP_BLE_MESH_FEATURE_PROXY

ESP_BLE_MESH_FEATURE_FRIEND

ESP_BLE_MESH_FEATURE_LOW_POWER

ESP_BLE_MESH_FEATURE_ALL_SUPPORTED

ESP_BLE_MESH_ADDR_IS_UNICAST(addr)

ESP_BLE_MESH_ADDR_IS_GROUP(addr)

ESP_BLE_MESH_ADDR_IS_VIRTUAL(addr)

ESP_BLE_MESH_ADDR_IS_RFU(addr)

ESP_BLE_MESH_INVALID_NODE_INDEX

ESP_BLE_MESH_TRANSMIT(count, int_ms)

Encode transmission count & interval steps.

Note: For example, ESP_BLE_MESH_TRANSMIT(2, 20) means that the message will be sent about 90ms(count is 3, step is 1, interval is 30 ms which includes 10ms of advertising interval random delay).

Parameters

- **count** –Number of retransmissions (first transmission is excluded).
- **int_ms** –Interval steps in milliseconds. Must be greater than 0 and a multiple of 10.

Returns BLE Mesh transmit value that can be used e.g. for the default values of the Configuration Model data.

ESP_BLE_MESH_GET_TRANSMIT_COUNT(transmit)

Decode transmit count from a transmit value.

Parameters
- **transmit** –Encoded transmit count & interval value.

Returns Transmission count (actual transmissions equal to N + 1).

ESP_BLE_MESH_GET_TRANSMIT_INTERVAL(transmit)

Decode transmit interval from a transmit value.

Parameters
- **transmit** –Encoded transmit count & interval value.

Returns Transmission interval in milliseconds.

ESP_BLE_MESH_PUBLISH_TRANSMIT(count, int_ms)

Encode Publish Retransmit count & interval steps.

Parameters
- **count** –Number of retransmissions (first transmission is excluded).
- **int_ms** –Interval steps in milliseconds. Must be greater than 0 and a multiple of 50.

Returns BLE Mesh transmit value that can be used e.g. for the default values of the Configuration Model data.

ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_COUNT(transmit)

Decode Publish Retransmit count from a given value.

Parameters
- **transmit** –Encoded Publish Retransmit count & interval value.

Returns Retransmission count (actual transmissions equal to N + 1).

ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_INTERVAL(transmit)

Decode Publish Retransmit interval from a given value.

Callbacks which are not needed to be initialized by users (set with 0 and will be initialized internally)

Parameters
- **transmit** –Encoded Publish Retransmit count & interval value.

Returns Transmission interval in milliseconds.

ESP_BLE_MESH_PROV_STATIC_OOB_MAX_LEN

Maximum length of string used by Output OOB authentication

ESP_BLE_MESH_PROV_OUTPUT_OOB_MAX_LEN

Maximum length of string used by Output OOB authentication

ESP_BLE_MESH_PROV_INPUT_OOB_MAX_LEN

Macros used to define message opcode

ESP_BLE_MESH_MODEL_OP_1(b0)

ESP_BLE_MESH_MODEL_OP_2(b0, b1)

ESP_BLE_MESH_MODEL_OP_3(b0, cid)

This macro is associated with BLE_MESH_MODEL_CB in mesh_access.h

ESP_BLE_MESH_SIG_MODEL(_id, _op, _pub, _user_data)

This macro is associated with BLE_MESH_MODEL_VND_CB in mesh_access.h

ESP_BLE_MESH_VENDOR_MODEL(_company, _id, _op, _pub, _user_data)

ESP_BLE_MESH_ELEMENT(_loc, _mods, _vnd_mods)

Helper to define a BLE Mesh element within an array.

In case the element has no SIG or Vendor models, the helper macro ESP_BLE_MESH_MODEL_NONE can be given instead.

Note: This macro is associated with BLE_MESH_ELEM in mesh_access.h

---

Parameters

- **_loc** –Location Descriptor.
- **_mods** –Array of SIG models.
- **_vnd_mods** –Array of vendor models.

ESP_BLE_MESH_PROV(uuid, sta_val, sta_val_len, out_size, out_act, in_size, in_act)

BT_OCTET32_LEN

BD_ADDR_LEN

ESP_BLE_MESH_ADDR_TYPE_PUBLIC

ESP_BLE_MESH_ADDR_TYPE_RANDOM

ESP_BLE_MESH_ADDR_TYPE_RPA_PUBLIC

ESP_BLE_MESH_ADDR_TYPE_RPA_RANDOM

ESP_BLE_MESH_MODEL_PUB_DEFINE(_name, _msg_len, _role)

Define a model publication context.

Parameters

- **_name** –Variable name given to the context.
- **_msg_len** –Length of the publication message.
- **_role** –Role of the device which contains the model.

ESP_BLE_MESH_MODEL_OP(_opcode, _min_len)

Define a model operation context.

Parameters

- **_opcode** –Message opcode.
- **_min_len** –Message minimum length.

ESP_BLE_MESH_MODEL_OP_END

Define the terminator for the model operation table. Each model operation struct array must use this terminator as the end tag of the operation unit.

ESP_BLE_MESH_MODEL_NONE

Helper to define an empty model array. This structure is associated with BLE_MESH_MODEL_NONE in mesh_access.h

ADD_DEV_RM_AFTER_PROV_FLAG

Device will be removed from queue after provisioned successfully

ADD_DEV_START_PROV_NOW_FLAG

Start provisioning device immediately

ADD_DEV_FLUSHABLE_DEV_FLAG

Device can be remove when queue is full and new device is going to added

DEL_DEV_ADDR_FLAG

DEL_DEV_UUID_FLAG

PROV_DATA_NET_IDX_FLAG

PROV_DATA_FLAGS_FLAG

PROV_DATA_IV_INDEX_FLAG

ESP_BLE_MESH_HEARTBEAT_FILTER_ACCEPTLIST

ESP_BLE_MESH_HEARTBEAT_FILTER_REJECTLIST

Provisioner heartbeat filter operation

ESP_BLE_MESH_HEARTBEAT_FILTER_ADD

ESP_BLE_MESH_HEARTBEAT_FILTER_REMOVE

ESP_BLE_MESH_MODEL_ID_CONFIG_SRV

BLE Mesh models related Model ID and Opcode definitions.

< Foundation Models

ESP_BLE_MESH_MODEL_ID_CONFIG_CLI

ESP_BLE_MESH_MODEL_ID_HEALTH_SRV

ESP_BLE_MESH_MODEL_ID_HEALTH_CLI

Models from the Mesh Model Specification

ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_SRV

ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_CLI

ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_CLI

ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV

ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI

ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_SRV

ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_CLI

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_CLI

ESP_BLE_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_USER_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_PROP_CLI

ESP_BLE_MESH_MODEL_ID_SENSOR_SRV

ESP_BLE_MESH_MODEL_ID_SENSOR_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_SENSOR_CLI

ESP_BLE_MESH_MODEL_ID_TIME_SRV

ESP_BLE_MESH_MODEL_ID_TIME_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_TIME_CLI

ESP_BLE_MESH_MODEL_ID_SCENE_SRV

ESP_BLE_MESH_MODEL_ID_SCENE_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_SCENE_CLI

ESP_BLE_MESH_MODEL_ID_SCHEDULER_SRV

ESP_BLE_MESH_MODEL_ID_SCHEDULER_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_SCHEDULER_CLI

ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI

ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_CLI

ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_CLI

ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_CLI

ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_LIGHT_LC_CLI

ESP_BLE_MESH_MODEL_OP_BEACON_GET
Config Beacon Get

ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET
Config Composition Data Get

ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET
Config Default TTL Get

ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET
Config GATT Proxy Get

ESP_BLE_MESH_MODEL_OP_RELAY_GET
Config Relay Get

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET
Config Model Publication Get

ESP_BLE_MESH_MODEL_OP_FRIEND_GET
Config Friend Get

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_GET
Config Heartbeat Publication Get

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_GET
Config Heartbeat Subscription Get

ESP_BLE_MESH_MODEL_OP_NET_KEY_GET
Config NetKey Get

ESP_BLE_MESH_MODEL_OP_APP_KEY_GET
Config AppKey Get

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET
Config Node Identity Get

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_GET
Config SIG Model Subscription Get

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET
Config Vendor Model Subscription Get

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_GET
Config SIG Model App Get

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET
Config Vendor Model App Get

ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_GET
Config Key Refresh Phase Get

ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_GET
Config Low Power Node PollTimeout Get

ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_GET
Config Network Transmit Get

ESP_BLE_MESH_MODEL_OP_BEACON_SET
Config Beacon Set

ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET
Config Default TTL Set

ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET
Config GATT Proxy Set

ESP_BLE_MESH_MODEL_OP_RELAY_SET
Config Relay Set

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET
Config Model Publication Set

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
Config Model Subscription Add

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD
Config Model Subscription Virtual Address Add

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE
Config Model Subscription Delete

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELETE
Config Model Subscription Virtual Address Delete

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE
Config Model Subscription Overwrite

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVERWRITE
Config Model Subscription Virtual Address Overwrite

ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD
Config NetKey Add

ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD
Config AppKey Add

ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND

Config Model App Bind

ESP_BLE_MESH_MODEL_OP_NODE_RESET

Config Node Reset

ESP_BLE_MESH_MODEL_OP_FRIEND_SET

Config Friend Set

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET

Config Heartbeat Publication Set

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET

Config Heartbeat Subscription Set

ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE

Config NetKey Update

ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE

Config NetKey Delete

ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE

Config AppKey Update

ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE

Config AppKey Delete

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET

Config Node Identity Set

ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_SET

Config Key Refresh Phase Set

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET

Config Model Publication Virtual Address Set

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL

Config Model Subscription Delete All

ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND

Config Model App Unbind

ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET

Config Network Transmit Set

ESP_BLE_MESH_MODEL_OP_BEACON_STATUS

ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_STATUS

ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_STATUS

ESP_BLE_MESH_MODEL_OP_GATT_PROXY_STATUS

ESP_BLE_MESH_MODEL_OP_RELAY_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_STATUS

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_LIST

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_LIST

ESP_BLE_MESH_MODEL_OP_NET_KEY_STATUS

ESP_BLE_MESH_MODEL_OP_NET_KEY_LIST

ESP_BLE_MESH_MODEL_OP_APP_KEY_STATUS

ESP_BLE_MESH_MODEL_OP_APP_KEY_LIST

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_APP_STATUS

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_LIST

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_LIST

ESP_BLE_MESH_MODEL_OP_NODE_RESET_STATUS

ESP_BLE_MESH_MODEL_OP_FRIEND_STATUS

ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_STATUS

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_STATUS

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_STATUS

ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_STATUS

ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_STATUS

ESP_BLE_MESH_CFG_STATUS_SUCCESS

ESP_BLE_MESH_CFG_STATUS_INVALID_ADDRESS

ESP_BLE_MESH_CFG_STATUS_INVALID_MODEL

ESP_BLE_MESH_CFG_STATUS_INVALID_APPKEY

ESP_BLE_MESH_CFG_STATUS_INVALID_NETKEY

ESP_BLE_MESH_CFG_STATUS_INSUFFICIENT_RESOURCES

ESP_BLE_MESH_CFG_STATUS_KEY_INDEX_ALREADY_STORED

ESP_BLE_MESH_CFG_STATUS_INVALID_PUBLISH_PARAMETERS

ESP_BLE_MESH_CFG_STATUS_NOT_A_SUBSCRIBE_MODEL

ESP_BLE_MESH_CFG_STATUS_STORAGE_FAILURE

ESP_BLE_MESH_CFG_STATUS_FEATURE_NOT_SUPPORTED

ESP_BLE_MESH_CFG_STATUS_CANNOT_UPDATE

ESP_BLE_MESH_CFG_STATUS_CANNOT_REMOVE

ESP_BLE_MESH_CFG_STATUS_CANNOT_BIND

ESP_BLE_MESH_CFG_STATUS_TEMP_UNABLE_TO_CHANGE_STATE

ESP_BLE_MESH_CFG_STATUS_CANNOT_SET

ESP_BLE_MESH_CFG_STATUS_UNSPECIFIED_ERROR

ESP_BLE_MESH_CFG_STATUS_INVALID_BINDING

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET
Health Fault Get

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET
Health Period Get

ESP_BLE_MESH_MODEL_OP_ATTENTION_GET
Health Attention Get

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR
Health Fault Clear

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK
Health Fault Clear Unacknowledged

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST
Health Fault Test

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK
Health Fault Test Unacknowledged

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET
Health Period Set

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK
Health Period Set Unacknowledged

ESP_BLE_MESH_MODEL_OP_ATTENTION_SET
Health Attention Set

ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK
Health Attention Set Unacknowledged

ESP_BLE_MESH_MODEL_OP_HEALTH_CURRENT_STATUS

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_STATUS

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_STATUS

ESP_BLE_MESH_MODEL_OP_ATTENTION_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_GET

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS
Generic Level Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_GET

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET

ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNACK

Generic Default Transition Time Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_GET

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS

Generic Power OnOff Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_GET

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS

Generic Power OnOff Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK

Generic Power Level Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_GET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_GET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_GET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_GET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS

Generic Power Level Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK

Generic Battery Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_GET

ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS

Generic Location Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_GET

ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_GET

ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS

Generic Location Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET

ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET

ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK

Generic Manufacturer Property Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_GET

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_GET

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_STATUS

Generic Admin Property Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_GET

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS
Generic User Property Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_GET

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS
Generic Client Property Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET

ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS
Sensor Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET

ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK

ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK

ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS

ESP_BLE_MESH_MODEL_OP_TIME_GET

ESP_BLE_MESH_MODEL_OP_TIME_SET

ESP_BLE_MESH_MODEL_OP_TIME_STATUS

ESP_BLE_MESH_MODEL_OP_TIME_ROLE_GET

ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET

ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS

ESP_BLE_MESH_MODEL_OP_TIME_ZONE_GET

ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET

ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS

ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_GET

ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_SET

ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_STATUS

Scene Message Opcode

ESP_BLE_MESH_MODEL_OP_SCENE_GET

ESP_BLE_MESH_MODEL_OP_SCENE_RECALL

ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNACK

ESP_BLE_MESH_MODEL_OP_SCENE_STATUS

ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_GET

ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS
Scene Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_SCENE_STORE

ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK

ESP_BLE_MESH_MODEL_OP_SCENE_DELETE

ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNACK
Scheduler Message Opcode

ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET

ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS

ESP_BLE_MESH_MODEL_OP_SCHEDULER_GET

ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS
Scheduler Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET

ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_STATUS

Light Lightness Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET_UNACK

Light CTL Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_STATUS
Light CTL Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET_UNACK
Light HSL Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_STATUS
Light HSL Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET_UNACK
Light xyL Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_STATUS
Light xyL Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET_UNACK
Light Control Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_STATUS

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS

ESP_BLE_MESH_MODEL_STATUS_SUCCESS

ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MIN

ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MAX

ESP_BLE_MESH_SERVER_RSP_BY_APP
Response need to be sent in the application

ESP_BLE_MESH_SERVER_AUTO_RSP
Response will be sent internally

Type Definitions

typedef uint8_t **esp_ble_mesh_octet16_t**[ESP_BLE_MESH_OCTET16_LEN]
Define the BLE Mesh octet 8 bytes size

typedef uint8_t **esp_ble_mesh_octet8_t**[ESP_BLE_MESH_OCTET8_LEN]
Invalid Company ID

typedef uint32_t **esp_ble_mesh_cb_t**

typedef uint8_t **UINT8**

typedef uint16_t **UINT16**

typedef uint32_t **UINT32**

typedef uint64_t **UINT64**

typedef *UINT8* **BT_OCTET32**[BT_OCTET32_LEN]

typedef uint8_t **BD_ADDR**[BD_ADDR_LEN]

typedef uint8_t **esp_ble_mesh_bd_addr_t**[BD_ADDR_LEN]

typedef uint8_t **esp_ble_mesh_addr_type_t**

BLE device address type.

typedef struct *esp_ble_mesh_model* **esp_ble_mesh_model_t**

typedef uint8_t **esp_ble_mesh_dev_add_flag_t**

typedef uint32_t **esp_ble_mesh_opcode_config_client_get_t**

esp_ble_mesh_opcode_config_client_get_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by esp_ble_mesh_config_client_get_state. The following opcodes will only be used in the esp_ble_mesh_config_client_get_state function.

typedef uint32_t **esp_ble_mesh_opcode_config_client_set_t**

esp_ble_mesh_opcode_config_client_set_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by esp_ble_mesh_config_client_set_state. The following opcodes will only be used in the esp_ble_mesh_config_client_set_state function.

typedef uint32_t **esp_ble_mesh_opcode_config_status_t**

esp_ble_mesh_opcode_config_status_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by the Config Model messages The following opcodes are used by the BLE Mesh Config Server Model internally to respond to the Config Client Model' s request messages.

typedef uint8_t **esp_ble_mesh_cfg_status_t**

This typedef is only used to indicate the status code contained in some of the Configuration Server Model status message.

typedef uint32_t **esp_ble_mesh_opcode_health_client_get_t**

esp_ble_mesh_opcode_health_client_get_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by esp_ble_mesh_health_client_get_state. The following opcodes will only be used in the esp_ble_mesh_health_client_get_state function.

typedef uint32_t **esp_ble_mesh_opcode_health_client_set_t**

esp_ble_mesh_opcode_health_client_set_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by esp_ble_mesh_health_client_set_state. The following opcodes will only be used in the esp_ble_mesh_health_client_set_state function.

typedef uint32_t **esp_ble_mesh_health_model_status_t**

esp_ble_mesh_health_model_status_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by the Health Model messages. The following opcodes are used by the BLE Mesh Health Server Model internally to respond to the Health Client Model' s request messages.

typedef uint32_t **esp_ble_mesh_generic_message_opcode_t**

esp_ble_mesh_generic_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_generic_client_get_state & esp_ble_mesh_generic_client_set_state. Generic OnOff Message Opcode

typedef uint32_t **esp_ble_mesh_sensor_message_opcode_t**

esp_ble_mesh_sensor_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_sensor_client_get_state & esp_ble_mesh_sensor_client_set_state. Sensor Message Opcode

typedef uint32_t **esp_ble_mesh_time_scene_message_opcode_t**

esp_ble_mesh_time_scene_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_time_scene_client_get_state & esp_ble_mesh_time_scene_client_set_state. Time Message Opcode

typedef uint32_t **esp_ble_mesh_light_message_opcode_t**

esp_ble_mesh_light_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_light_client_get_state & esp_ble_mesh_light_client_set_state. Light Lightness Message Opcode

typedef uint32_t **esp_ble_mesh_opcode_t**

End of defines of esp_ble_mesh_opcode_t

typedef uint8_t **esp_ble_mesh_model_status_t**

This typedef is only used to indicate the status code contained in some of the server models (e.g. Generic Server Model) status message.

Enumerations

enum **esp_ble_mesh_cb_type_t**

Values:

enumerator **ESP_BLE_MESH_TYPE_PROV_CB**

enumerator **ESP_BLE_MESH_TYPE_OUTPUT_NUM_CB**

enumerator **ESP_BLE_MESH_TYPE_OUTPUT_STR_CB**

enumerator **ESP_BLE_MESH_TYPE_INTPUT_CB**

enumerator **ESP_BLE_MESH_TYPE_LINK_OPEN_CB**

enumerator **ESP_BLE_MESH_TYPE_LINK_CLOSE_CB**

enumerator **ESP_BLE_MESH_TYPE_COMPLETE_CB**

enumerator **ESP_BLE_MESH_TYPE_RESET_CB**

enum **esp_ble_mesh_oob_method_t**

Values:

enumerator **ESP_BLE_MESH_NO_OOB**

enumerator **ESP_BLE_MESH_STATIC_OOB**

enumerator **ESP_BLE_MESH_OUTPUT_OOB**

enumerator **ESP_BLE_MESH_INPUT_OOB**

enum **esp_ble_mesh_output_action_t**

Values:

enumerator **ESP_BLE_MESH_NO_OUTPUT**

enumerator **ESP_BLE_MESH_BLINK**

enumerator **ESP_BLE_MESH_BEEP**

enumerator **ESP_BLE_MESH_VIBRATE**

enumerator **ESP_BLE_MESH_DISPLAY_NUMBER**

enumerator **ESP_BLE_MESH_DISPLAY_STRING**

enum **esp_ble_mesh_input_action_t**

Values:

enumerator **ESP_BLE_MESH_NO_INPUT**

enumerator **ESP_BLE_MESH_PUSH**

enumerator **ESP_BLE_MESH_TWIST**

enumerator **ESP_BLE_MESH_ENTER_NUMBER**

enumerator **ESP_BLE_MESH_ENTER_STRING**

enum **esp_ble_mesh_prov_bearer_t**

Values:

enumerator **ESP_BLE_MESH_PROV_ADV**

enumerator **ESP_BLE_MESH_PROV_GATT**

enum **esp_ble_mesh_prov_oob_info_t**

Values:

enumerator **ESP_BLE_MESH_PROV_OOB_OTHER**

enumerator **ESP_BLE_MESH_PROV_OOB_URI**

enumerator **ESP_BLE_MESH_PROV_OOB_2D_CODE**

enumerator **ESP_BLE_MESH_PROV_OOB_BAR_CODE**

enumerator **ESP_BLE_MESH_PROV_OOB_NFC**

enumerator **ESP_BLE_MESH_PROV_OOB_NUMBER**

enumerator **ESP_BLE_MESH_PROV_OOB_STRING**

enumerator **ESP_BLE_MESH_PROV_OOB_ON_BOX**

enumerator **ESP_BLE_MESH_PROV_OOB_IN_BOX**

enumerator **ESP_BLE_MESH_PROV_OOB_ON_PAPER**

enumerator **ESP_BLE_MESH_PROV_OOB_IN_MANUAL**

enumerator **ESP_BLE_MESH_PROV_OOB_ON_DEV**

enum **esp_ble_mesh_dev_role_t**

Values:

enumerator **ROLE_NODE**

enumerator **ROLE_PROVISIONER**

enumerator **ROLE_FAST_PROV**

enum **esp_ble_mesh_fast_prov_action_t**

Values:

enumerator **FAST_PROV_ACT_NONE**

enumerator **FAST_PROV_ACT_ENTER**

enumerator **FAST_PROV_ACT_SUSPEND**

enumerator **FAST_PROV_ACT_EXIT**

enumerator **FAST_PROV_ACT_MAX**

enum **esp_ble_mesh_proxy_filter_type_t**

Values:

enumerator **PROXY_FILTER_WHITELIST**

enumerator **PROXY_FILTER_BLACKLIST**

enum **esp_ble_mesh_prov_cb_event_t**

Values:

enumerator **ESP_BLE_MESH_PROV_REGISTER_COMP_EVT**

Initialize BLE Mesh provisioning capabilities and internal data information completion event

enumerator **ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT**

Set the unprovisioned device name completion event

enumerator **ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT**

Enable node provisioning functionality completion event

enumerator **ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT**

Disable node provisioning functionality completion event

enumerator **ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT**

Establish a BLE Mesh link event

enumerator **ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT**

Close a BLE Mesh link event

enumerator **ESP_BLE_MESH_NODE_PROV_OOB_PUB_KEY_EVT**

Generate Node input OOB public key event

enumerator **ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT**

Generate Node Output Number event

enumerator **ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT**

Generate Node Output String event

enumerator **ESP_BLE_MESH_NODE_PROV_INPUT_EVT**

Event requiring the user to input a number or string

enumerator **ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT**

Provisioning done event

enumerator **ESP_BLE_MESH_NODE_PROV_RESET_EVT**

Provisioning reset event

enumerator **ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT**

Node set oob public key completion event

enumerator **ESP_BLE_MESH_NODE_PROV_INPUT_NUMBER_COMP_EVT**

Node input number completion event

enumerator **ESP_BLE_MESH_NODE_PROV_INPUT_STRING_COMP_EVT**

Node input string completion event

enumerator **ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT**

Enable BLE Mesh Proxy Identity advertising completion event

enumerator **ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT**

Enable BLE Mesh GATT Proxy Service completion event

enumerator **ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT**

Disable BLE Mesh GATT Proxy Service completion event

enumerator **ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT**

Node add NetKey locally completion event

enumerator **ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT**

Node add AppKey locally completion event

enumerator **ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT**

Node bind AppKey to model locally completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT**

Provisioner enable provisioning functionality completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT**

Provisioner disable provisioning functionality completion event

enumerator **ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT**

Provisioner receives unprovisioned device beacon event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT**

Provisioner read unprovisioned device OOB public key event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT**

Provisioner input value for provisioning procedure event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT**

Provisioner output value for provisioning procedure event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT**

Provisioner establish a BLE Mesh link event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT**

Provisioner close a BLE Mesh link event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT**

Provisioner provisioning done event

enumerator **ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT**

Provisioner add a device to the list which contains devices that are waiting/going to be provisioned completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT**

Provisioner start to provision an unprovisioned device completion event

enumerator **ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT**

Provisioner delete a device from the list, close provisioning link with the device completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT**

Provisioner set the value to be compared with part of the unprovisioned device UUID completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT**

Provisioner set net_idx/flags/iv_index used for provisioning completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT**

Provisioner set static oob value used for provisioning completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT**

Provisioner set unicast address of primary element completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT**

Provisioner read unprovisioned device OOB public key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT**

Provisioner input number completion event

enumerator **ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT**

Provisioner input string completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT**

Provisioner set node name completion event

enumerator **ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT**
Provisioner add local app key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT**
Provisioner update local app key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT**
Provisioner bind local model with local app key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT**
Provisioner add local network key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT**
Provisioner update local network key completion event

enumerator **ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT**
Provisioner store node composition data completion event

enumerator **ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT**
Provisioner delete node with uuid completion event

enumerator **ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT**
Provisioner delete node with unicast address completion event

enumerator **ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT**
Provisioner start to receive heartbeat message completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT**
Provisioner set the heartbeat filter type completion event

enumerator **ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT**
Provisioner set the heartbeat filter information completion event

enumerator **ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT**
Provisioner receive heartbeat message event

enumerator **ESP_BLE_MESH_PROVISIONER_DIRECT_ERASE_SETTINGS_COMP_EVT**
Provisioner directly erase settings completion event

enumerator **ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT**
Provisioner open settings with index completion event

enumerator **ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT**
Provisioner open settings with user id completion event

enumerator **ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT**
Provisioner close settings with index completion event

enumerator **ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT**

Provisioner close settings with user id completion event

enumerator **ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT**

Provisioner delete settings with index completion event

enumerator **ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT**

Provisioner delete settings with user id completion event

enumerator **ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT**

Set fast provisioning information (e.g. unicast address range, net_idx, etc.) completion event

enumerator **ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT**

Set fast provisioning action completion event

enumerator **ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT**

Receive Heartbeat message event

enumerator **ESP_BLE_MESH_LPN_ENABLE_COMP_EVT**

Enable Low Power Node completion event

enumerator **ESP_BLE_MESH_LPN_DISABLE_COMP_EVT**

Disable Low Power Node completion event

enumerator **ESP_BLE_MESH_LPN_POLL_COMP_EVT**

Low Power Node send Friend Poll completion event

enumerator **ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT**

Low Power Node establishes friendship event

enumerator **ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT**

Low Power Node terminates friendship event

enumerator **ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT**

Friend Node establishes friendship event

enumerator **ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT**

Friend Node terminates friendship event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT**

Proxy Client receives Network ID advertising packet event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT**

Proxy Client establishes connection successfully event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT**

Proxy Client terminates connection successfully event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT**

Proxy Client receives Proxy Filter Status event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT**

Proxy Client connect completion event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT**

Proxy Client disconnect completion event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT**

Proxy Client set filter type completion event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT**

Proxy Client add filter address completion event

enumerator **ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT**

Proxy Client remove filter address completion event

enumerator **ESP_BLE_MESH_PROXY_SERVER_CONNECTED_EVT**

Proxy Server establishes connection successfully event

enumerator **ESP_BLE_MESH_PROXY_SERVER_DISCONNECTED_EVT**

Proxy Server terminates connection successfully event

enumerator **ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT**

Local model subscribes group address completion event

enumerator **ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT**

Local model unsubscribes group address completion event

enumerator **ESP_BLE_MESH_DEINIT_MESH_COMP_EVT**

De-initialize BLE Mesh stack completion event

enumerator **ESP_BLE_MESH_PROV_EVT_MAX**

enum **[anonymous]**

BLE Mesh server models related definitions.

This enum value is the flag of transition timer operation

Values:

enumerator **ESP_BLE_MESH_SERVER_TRANS_TIMER_START**

enumerator **ESP_BLE_MESH_SERVER_FLAG_MAX**

enum **esp_ble_mesh_server_state_type_t**

This enum value is the type of server model states

Values:

enumerator **ESP_BLE_MESH_GENERIC_ONOFF_STATE**

enumerator **ESP_BLE_MESH_GENERIC_LEVEL_STATE**

enumerator **ESP_BLE_MESH_GENERIC_ONPOWERUP_STATE**

enumerator **ESP_BLE_MESH_GENERIC_POWER_ACTUAL_STATE**

enumerator **ESP_BLE_MESH_LIGHT_LIGHTNESS_ACTUAL_STATE**

enumerator **ESP_BLE_MESH_LIGHT_LIGHTNESS_LINEAR_STATE**

enumerator **ESP_BLE_MESH_LIGHT_CTL_LIGHTNESS_STATE**

enumerator **ESP_BLE_MESH_LIGHT_CTL_TEMP_DELTA_UV_STATE**

enumerator **ESP_BLE_MESH_LIGHT_HSL_STATE**

enumerator **ESP_BLE_MESH_LIGHT_HSL_LIGHTNESS_STATE**

enumerator **ESP_BLE_MESH_LIGHT_HSL_HUE_STATE**

enumerator **ESP_BLE_MESH_LIGHT_HSL_SATURATION_STATE**

enumerator **ESP_BLE_MESH_LIGHT_XYL_LIGHTNESS_STATE**

enumerator **ESP_BLE_MESH_LIGHT_LC_LIGHT_ONOFF_STATE**

enumerator **ESP_BLE_MESH_SERVER_MODEL_STATE_MAX**

enum **esp_ble_mesh_model_cb_event_t**

Values:

enumerator **ESP_BLE_MESH_MODEL_OPERATION_EVT**

User-defined models receive messages from peer devices (e.g. get, set, status, etc) event

enumerator **ESP_BLE_MESH_MODEL_SEND_COMP_EVT**

User-defined models send messages completion event

enumerator **ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT**

User-defined models publish messages completion event

enumerator **ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT**

User-defined client models receive publish messages event

enumerator **ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT**

Timeout event for the user-defined client models that failed to receive response from peer server models

enumerator **ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT**

When a model is configured to publish messages periodically, this event will occur during every publish period

enumerator **ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT**

Server models update state value completion event

enumerator **ESP_BLE_MESH_MODEL_EVT_MAX**

ESP-BLE-MESH Core API Reference

This section contains ESP-BLE-MESH Core related APIs, which can be used to initialize ESP-BLE-MESH stack, provision, send/publish messages, etc.

This API reference covers six components:

- *ESP-BLE-MESH Stack Initialization*
- *Reading of Local Data Information*
- *Low Power Operation (Updating)*
- *Send/Publish Messages, add Local AppKey, etc.*
- *ESP-BLE-MESH Node/Provisioner Provisioning*
- *ESP-BLE-MESH GATT Proxy Server*

ESP-BLE-MESH Stack Initialization

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_common_api.h

Functions

esp_err_t **esp_ble_mesh_init**(*esp_ble_mesh_prov_t* *prov, *esp_ble_mesh_comp_t* *comp)

Initialize BLE Mesh module. This API initializes provisioning capabilities and composition data information.

Note: After calling this API, the device needs to call esp_ble_mesh_prov_enable() to enable provisioning functionality again.

Parameters

- **prov** –**[in]** Pointer to the device provisioning capabilities. This pointer must remain valid during the lifetime of the BLE Mesh device.
- **comp** –**[in]** Pointer to the device composition data information. This pointer must remain valid during the lifetime of the BLE Mesh device.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_deinit**(*esp_ble_mesh_deinit_param_t* *param)

De-initialize BLE Mesh module.

Note: This function shall be invoked after esp_ble_mesh_client_model_deinit().

Parameters **param** –**[in]** Pointer to the structure of BLE Mesh deinit parameters.

Returns ESP_OK on success or error code otherwise.

Reading of Local Data Information

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_local_data_operation_api.h

Functions

int32_t **esp_ble_mesh_get_model_publish_period**(*esp_ble_mesh_model_t* *model)

Get the model publish period, the unit is ms.

Parameters **model** –**[in]** Model instance pointer.
Returns Publish period value on success, 0 or (negative) error code from errno.h on failure.

uint16_t **esp_ble_mesh_get_primary_element_address**(void)

Get the address of the primary element.

Returns Address of the primary element on success, or ESP_BLE_MESH_ADDR_UNASSIGNED on failure which means the device has not been provisioned.

uint16_t ***esp_ble_mesh_is_model_subscribed_to_group**(*esp_ble_mesh_model_t* *model, uint16_t group_addr)

Check if the model has subscribed to the given group address. Note: E.g., once a status message is received and the destination address is a group address, the model uses this API to check if it is successfully subscribed to the given group address.

Parameters
- **model** –**[in]** Pointer to the model.
- **group_addr** –**[in]** Group address.

Returns Pointer to the group address within the Subscription List of the model on success, or NULL on failure which means the model has not subscribed to the given group address. Note: With the pointer to the group address returned, you can reset the group address to 0x0000 in order to unsubscribe the model from the group.

esp_ble_mesh_elem_t ***esp_ble_mesh_find_element**(uint16_t element_addr)

Find the BLE Mesh element pointer via the element address.

Parameters **element_addr** –**[in]** Element address.
Returns Pointer to the element on success, or NULL on failure.

uint8_t **esp_ble_mesh_get_element_count**(void)

Get the number of elements that have been registered.

Returns Number of elements.

esp_ble_mesh_model_t ***esp_ble_mesh_find_vendor_model**(const *esp_ble_mesh_elem_t* *element, uint16_t company_id, uint16_t model_id)

Find the Vendor specific model with the given element, the company ID and the Vendor Model ID.

Parameters
- **element** –**[in]** Element to which the model belongs.
- **company_id** –**[in]** A 16-bit company identifier assigned by the Bluetooth SIG.
- **model_id** –**[in]** A 16-bit vendor-assigned model identifier.

Returns Pointer to the Vendor Model on success, or NULL on failure which means the Vendor Model is not found.

esp_ble_mesh_model_t ***esp_ble_mesh_find_sig_model**(const *esp_ble_mesh_elem_t* *element, uint16_t model_id)

Find the SIG model with the given element and Model id.

Parameters
- **element** –**[in]** Element to which the model belongs.

- **model_id** –**[in]** SIG model identifier.

Returns Pointer to the SIG Model on success, or NULL on failure which means the SIG Model is not found.

const *esp_ble_mesh_comp_t* ***esp_ble_mesh_get_composition_data**(void)

Get the Composition data which has been registered.

Returns Pointer to the Composition data on success, or NULL on failure which means the Composition data is not initialized.

esp_err_t **esp_ble_mesh_model_subscribe_group_addr**(uint16_t element_addr, uint16_t company_id, uint16_t model_id, uint16_t group_addr)

A local model of node or Provisioner subscribes a group address.

Note: This function shall not be invoked before node is provisioned or Provisioner is enabled.

Parameters

- **element_addr** –**[in]** Unicast address of the element to which the model belongs.
- **company_id** –**[in]** A 16-bit company identifier.
- **model_id** –**[in]** A 16-bit model identifier.
- **group_addr** –**[in]** The group address to be subscribed.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_model_unsubscribe_group_addr**(uint16_t element_addr, uint16_t company_id, uint16_t model_id, uint16_t group_addr)

A local model of node or Provisioner unsubscribes a group address.

Note: This function shall not be invoked before node is provisioned or Provisioner is enabled.

Parameters

- **element_addr** –**[in]** Unicast address of the element to which the model belongs.
- **company_id** –**[in]** A 16-bit company identifier.
- **model_id** –**[in]** A 16-bit model identifier.
- **group_addr** –**[in]** The subscribed group address.

Returns ESP_OK on success or error code otherwise.

const uint8_t ***esp_ble_mesh_node_get_local_net_key**(uint16_t net_idx)

This function is called by Node to get the local NetKey.

Parameters **net_idx** –**[in]** NetKey index.
Returns NetKey on success, or NULL on failure.

const uint8_t ***esp_ble_mesh_node_get_local_app_key**(uint16_t app_idx)

This function is called by Node to get the local AppKey.

Parameters **app_idx** –**[in]** AppKey index.
Returns AppKey on success, or NULL on failure.

esp_err_t **esp_ble_mesh_node_add_local_net_key**(const uint8_t net_key[16], uint16_t net_idx)

This function is called by Node to add a local NetKey.

Note: This function can only be called after the device is provisioned.

Parameters

- **net_key** –**[in]** NetKey to be added.
- **net_idx** –**[in]** NetKey Index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_add_local_app_key**(const uint8_t app_key[16], uint16_t net_idx, uint16_t app_idx)

This function is called by Node to add a local AppKey.

Note: The net_idx must be an existing one. This function can only be called after the device is provisioned.

Parameters

- **app_key** –**[in]** AppKey to be added.
- **net_idx** –**[in]** NetKey Index.
- **app_idx** –**[in]** AppKey Index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_bind_app_key_to_local_model**(uint16_t element_addr, uint16_t company_id, uint16_t model_id, uint16_t app_idx)

This function is called by Node to bind AppKey to model locally.

Note: If going to bind app_key with local vendor model, the company_id shall be set to 0xFFFF. This function can only be called after the device is provisioned.

Parameters

- **element_addr** –**[in]** Node local element address
- **company_id** –**[in]** Node local company id
- **model_id** –**[in]** Node local model id
- **app_idx** –**[in]** Node local appkey index

Returns ESP_OK on success or error code otherwise.

Low Power Operation (Updating)

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_low_power_api.h

Functions

esp_err_t **esp_ble_mesh_lpn_enable**(void)

Enable BLE Mesh device LPN functionality.

Note: This API enables LPN functionality. Once called, the proper Friend Request will be sent.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_lpn_disable**(bool force)

Disable BLE Mesh device LPN functionality.

Parameters **force** –**[in]** when disabling LPN functionality, use this flag to indicate whether directly clear corresponding information or just send friend clear to disable it if friendship has already been established.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_lpn_poll** (void)

LPN tries to poll messages from the Friend Node.

Note: The Friend Poll message is sent by a Low Power node to ask the Friend node to send a message that it has stored for the Low Power node. Users can call this API to send Friend Poll message manually. If this API is not invoked, the bottom layer of the Low Power node will send Friend Poll before the PollTimeout timer expires. If the corresponding Friend Update is received and MD is set to 0, which means there are no messages for the Low Power node, then the Low Power node will stop scanning.

Returns ESP_OK on success or error code otherwise.

Send/Publish Messages, add Local AppKey, etc.

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_networking_api.h

Functions

esp_err_t **esp_ble_mesh_register_custom_model_callback** (*esp_ble_mesh_model_cb_t* callback)

Register BLE Mesh callback for user-defined models' operations. This callback can report the following events generated for the user-defined models:

- Call back the messages received by user-defined client and server models to the application layer;
- If users call esp_ble_mesh_server/client_model_send, this callback notifies the application layer of the send_complete event;
- If user-defined client model sends a message that requires response, and the response message is received after the timer expires, the response message will be reported to the application layer as published by a peer device;
- If the user-defined client model fails to receive the response message during a specified period of time, a timeout event will be reported to the application layer.

Note: The client models (i.e. Config Client model, Health Client model, Generic Client models, Sensor Client model, Scene Client model and Lighting Client models) that have been realized internally have their specific register functions. For example, esp_ble_mesh_register_config_client_callback is the register function for Config Client Model.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_model_msg_opcode_init** (uint8_t *data, uint32_t opcode)

Add the message opcode to the beginning of the model message before sending or publishing the model message.

Note: This API is only used to set the opcode of the message.

Parameters

- **data** –**[in]** Pointer to the message data.
- **opcode** –**[in]** The message opcode.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_client_model_init** (*esp_ble_mesh_model_t* *model)

Initialize the user-defined client model. All user-defined client models shall call this function to initialize the client model internal data. Node: Before calling this API, the op_pair_size and op_pair variabled within the user_data(defined using esp_ble_mesh_client_t_) of the client model need to be initialized.

Parameters **model** –**[in]** BLE Mesh Client model to which the message belongs.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_client_model_deinit** (*esp_ble_mesh_model_t* *model)

De-initialize the user-defined client model.

Note: This function shall be invoked before esp_ble_mesh_deinit() is called.

Parameters **model** –**[in]** Pointer of the Client model.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_server_model_send_msg** (*esp_ble_mesh_model_t* *model, *esp_ble_mesh_msg_ctx_t* *ctx, uint32_t opcode, uint16_t length, uint8_t *data)

Send server model messages(such as server model status messages).

Parameters
- **model** –**[in]** BLE Mesh Server Model to which the message belongs.
- **ctx** –**[in]** Message context, includes keys, TTL, etc.
- **opcode** –**[in]** Message opcode.
- **length** –**[in]** Message length (exclude the message opcode).
- **data** –**[in]** Parameters of Access Payload (exclude the message opcode) to be sent.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_client_model_send_msg** (*esp_ble_mesh_model_t* *model, *esp_ble_mesh_msg_ctx_t* *ctx, uint32_t opcode, uint16_t length, uint8_t *data, int32_t msg_timeout, bool need_rsp, *esp_ble_mesh_dev_role_t* device_role)

Send client model message (such as model get, set, etc).

Parameters
- **model** –**[in]** BLE Mesh Client Model to which the message belongs.
- **ctx** –**[in]** Message context, includes keys, TTL, etc.
- **opcode** –**[in]** Message opcode.
- **length** –**[in]** Message length (exclude the message opcode).
- **data** –**[in]** Parameters of the Access Payload (exclude the message opcode) to be sent.
- **msg_timeout** –**[in]** Time to get response to the message (in milliseconds).
- **need_rsp** –**[in]** TRUE if the opcode requires the peer device to reply, FALSE otherwise.
- **device_role** –**[in]** Role of the device (Node/Provisioner) that sends the message.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_model_publish** (*esp_ble_mesh_model_t* *model, uint32_t opcode, uint16_t length, uint8_t *data, *esp_ble_mesh_dev_role_t* device_role)

Send a model publication message.

Note: Before calling this function, the user needs to ensure that the model publication message (*esp_ble_mesh_model_pub_t::msg*) contains a valid message to be sent. And if users want to update the publishing message, this API should be called in ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT with the message updated.

Parameters

- **model** –**[in]** Mesh (client) Model publishing the message.
- **opcode** –**[in]** Message opcode.
- **length** –**[in]** Message length (exclude the message opcode).
- **data** –**[in]** Parameters of the Access Payload (exclude the message opcode) to be sent.
- **device_role** –**[in]** Role of the device (node/provisioner) publishing the message of the type esp_ble_mesh_dev_role_t.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_server_model_update_state**(*esp_ble_mesh_model_t* *model, *esp_ble_mesh_server_state_type_t* type, *esp_ble_mesh_server_state_value_t* *value)

Update a server model state value. If the model publication state is set properly (e.g. publish address is set to a valid address), it will publish corresponding status message.

Note: Currently this API is used to update bound state value, not for all server model states.

Parameters

- **model** –**[in]** Server model which is going to update the state.
- **type** –**[in]** Server model state type.
- **value** –**[in]** Server model state value.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_local_reset**(void)

Reset the provisioning procedure of the local BLE Mesh node.

Note: All provisioning information in this node will be deleted and the node needs to be reprovisioned. The API function esp_ble_mesh_node_prov_enable() needs to be called to start a new provisioning procedure.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_node_name**(uint16_t index, const char *name)

This function is called to set the node (provisioned device) name.

Note: index is obtained from the parameters of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Parameters

- **index** –**[in]** Index of the node in the node queue.
- **name** –**[in]** Name (end by '\0') to be set for the node.

Returns ESP_OK on success or error code otherwise.

const char ***esp_ble_mesh_provisioner_get_node_name**(uint16_t index)

This function is called to get the node (provisioned device) name.

Note: index is obtained from the parameters of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Parameters **index** –**[in]** Index of the node in the node queue.

Returns Node name on success, or NULL on failure.

uint16_t **esp_ble_mesh_provisioner_get_node_index**(const char *name)

This function is called to get the node (provisioned device) index.

Parameters **name** –**[in]** Name of the node (end by '\0').
Returns Node index on success, or an invalid value (0xFFFF) on failure.

esp_err_t **esp_ble_mesh_provisioner_store_node_comp_data**(uint16_t unicast_addr, uint8_t *data, uint16_t length)

This function is called to store the Composition Data of the node.

Parameters
- **unicast_addr** –**[in]** Element address of the node
- **data** –**[in]** Pointer of Composition Data
- **length** –**[in]** Length of Composition Data

Returns ESP_OK on success or error code otherwise.

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_uuid**(const uint8_t uuid[16])

This function is called to get the provisioned node information with the node device uuid.

Parameters **uuid** –**[in]** Device UUID of the node
Returns Pointer of the node info struct or NULL on failure.

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_addr**(uint16_t unicast_addr)

This function is called to get the provisioned node information with the node unicast address.

Parameters **unicast_addr** –**[in]** Unicast address of the node
Returns Pointer of the node info struct or NULL on failure.

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_name**(const char *name)

This function is called to get the provisioned node information with the node name.

Parameters **name** –**[in]** Name of the node (end by '\0').
Returns Pointer of the node info struct or NULL on failure.

uint16_t **esp_ble_mesh_provisioner_get_prov_node_count**(void)

This function is called by Provisioner to get provisioned node count.

Returns Number of the provisioned nodes.

const *esp_ble_mesh_node_t* ****esp_ble_mesh_provisioner_get_node_table_entry**(void)

This function is called by Provisioner to get the entry of the node table.

Note: After invoking the function to get the entry of nodes, users can use the "for" loop combined with the macro CONFIG_BLE_MESH_MAX_PROV_NODES to get each node's information. Before trying to read the node's information, users need to check if the node exists, i.e. if the *(*esp_ble_mesh_node_t* **node) is NULL. For example: ``` const *esp_ble_mesh_node_t* **entry = esp_ble_mesh_provisioner_get_node_table_entry(); for (int i = 0; i < CONFIG_BLE_MESH_MAX_PROV_NODES; i++) { const *esp_ble_mesh_node_t* *node = entry[i]; if (node) { ……} } ```

Returns Pointer to the start of the node table.

esp_err_t **esp_ble_mesh_provisioner_delete_node_with_uuid**(const uint8_t uuid[16])

This function is called to delete the provisioned node information with the node device uuid.

Parameters **uuid** –**[in]** Device UUID of the node
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_delete_node_with_addr**(uint16_t unicast_addr)

This function is called to delete the provisioned node information with the node unicast address.

Parameters **unicast_addr** –**[in]** Unicast address of the node

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_add_local_app_key**(const uint8_t app_key[16], uint16_t net_idx, uint16_t app_idx)

This function is called to add a local AppKey for Provisioner.

Note: app_key: If set to NULL, app_key will be generated internally. net_idx: Should be an existing one. app_idx: If it is going to be generated internally, it should be set to 0xFFFF, and the new app_idx will be reported via an event.

Parameters

- **app_key** –**[in]** The app key to be set for the local BLE Mesh stack.
- **net_idx** –**[in]** The network key index.
- **app_idx** –**[in]** The app key index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_update_local_app_key**(const uint8_t app_key[16], uint16_t net_idx, uint16_t app_idx)

This function is used to update a local AppKey for Provisioner.

Parameters

- **app_key** –**[in]** Value of the AppKey.
- **net_idx** –**[in]** Corresponding NetKey Index.
- **app_idx** –**[in]** The AppKey Index

Returns ESP_OK on success or error code otherwise.

const uint8_t ***esp_ble_mesh_provisioner_get_local_app_key**(uint16_t net_idx, uint16_t app_idx)

This function is called by Provisioner to get the local app key value.

Parameters

- **net_idx** –**[in]** Network key index.
- **app_idx** –**[in]** Application key index.

Returns App key on success, or NULL on failure.

esp_err_t **esp_ble_mesh_provisioner_bind_app_key_to_local_model**(uint16_t element_addr, uint16_t app_idx, uint16_t model_id, uint16_t company_id)

This function is called by Provisioner to bind own model with proper app key.

Note: company_id: If going to bind app_key with local vendor model, company_id should be set to 0xFFFF.

Parameters

- **element_addr** –**[in]** Provisioner local element address
- **app_idx** –**[in]** Provisioner local appkey index
- **model_id** –**[in]** Provisioner local model id
- **company_id** –**[in]** Provisioner local company id

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_add_local_net_key**(const uint8_t net_key[16], uint16_t net_idx)

This function is called by Provisioner to add local network key.

Note: net_key: If set to NULL, net_key will be generated internally. net_idx: If it is going to be generated internally, it should be set to 0xFFFF, and the new net_idx will be reported via an event.

Parameters
- **net_key** –**[in]** The network key to be added to the Provisioner local BLE Mesh stack.
- **net_idx** –**[in]** The network key index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_update_local_net_key**(const uint8_t net_key[16], uint16_t net_idx)

This function is called by Provisioner to update a local network key.

Parameters
- **net_key** –**[in]** Value of the NetKey.
- **net_idx** –**[in]** The NetKey Index.

Returns ESP_OK on success or error code otherwise.

const uint8_t ***esp_ble_mesh_provisioner_get_local_net_key**(uint16_t net_idx)

This function is called by Provisioner to get the local network key value.

Parameters **net_idx** –**[in]** Network key index.
Returns Network key on success, or NULL on failure.

esp_err_t **esp_ble_mesh_provisioner_recv_heartbeat**(bool enable)

This function is called by Provisioner to enable or disable receiving heartbeat messages.

Note: If enabling receiving heartbeat message successfully, the filter will be an empty rejectlist by default, which means all heartbeat messages received by the Provisioner will be reported to the application layer.

Parameters **enable** –**[in]** Enable or disable receiving heartbeat messages.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_heartbeat_filter_type**(uint8_t type)

This function is called by Provisioner to set the heartbeat filter type.

Note: 1. If the filter type is not the same with the current value, then all the filter entries will be cleaned.

a. If the previous type is rejectlist, and changed to acceptlist, then the filter will be an empty acceptlist, which means no heartbeat messages will be reported. Users need to add SRC or DST into the filter entry, then heartbeat messages from the SRC or to the DST will be reported.

Parameters **type** –**[in]** Heartbeat filter type (acceptlist or rejectlist).
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_heartbeat_filter_info**(uint8_t op, *esp_ble_mesh_heartbeat_filter_info_t* *info)

This function is called by Provisioner to add or remove a heartbeat filter entry.

a. If the operation is "REMOVE", the "hb_src" can be set to the SRC (can only be a unicast address) of heartbeat messages, and the "hb_dst" can be set to the DST (unicast address or group address), at least one of them needs to be set.
   - The filter entry with the same SRC or DST will be removed.

Note: 1. If the operation is "ADD", the "hb_src" can be set to the SRC (can only be a unicast address) of heartbeat messages, and the "hb_dst" can be set to the DST (unicast address or group address), at least one of them needs to be set.

- If only one of them is set, the filter entry will only use the configured SRC or DST to filter heartbeat messages.
- If both of them are set, the SRC and DST will both be used to decide if a heartbeat message will be handled.
- If SRC or DST already exists in some filter entry, then the corresponding entry will be cleaned firstly, then a new entry will be allocated to store the information.

Parameters
- **`op`** –**[in]** Add or REMOVE
- **`info`** –**[in]** Heartbeat filter entry information, including: hb_src - Heartbeat source address; hb_dst - Heartbeat destination address;

Returns ESP_OK on success or error code otherwise.

esp_err_t **`esp_ble_mesh_provisioner_direct_erase_settings`**(void)

This function is called by Provisioner to directly erase the mesh information from nvs namespace.

Note: This function can be invoked when the mesh stack is not initialized or has been de-initialized.

Returns ESP_OK on success or error code otherwise.

esp_err_t **`esp_ble_mesh_provisioner_open_settings_with_index`**(uint8_t index)

This function is called by Provisioner to open a nvs namespace for storing mesh information.

Note: Before open another nvs namespace, the previously opened nvs namespace must be closed firstly.

Parameters **`index`** –**[in]** Settings index.
Returns ESP_OK on success or error code otherwise.

esp_err_t **`esp_ble_mesh_provisioner_open_settings_with_uid`**(const char *uid)

This function is called by Provisioner to open a nvs namespace for storing mesh information.

Note: Before open another nvs namespace, the previously opened nvs namespace must be closed firstly.

Parameters **`uid`** –**[in]** Settings user id.
Returns ESP_OK on success or error code otherwise.

esp_err_t **`esp_ble_mesh_provisioner_close_settings_with_index`**(uint8_t index, bool erase)

This function is called by Provisioner to close a nvs namespace which is opened previously for storing mesh information.

Note: 1. Before closing the nvs namespace, it must be open.

a. When the function is invoked, the Provisioner functionality will be disabled firstly, and: a) If the "erase" flag is set to false, the mesh information will be cleaned (e.g. removing NetKey, AppKey, nodes, etc) from the mesh stack. b) If the "erase" flag is set to true, the mesh information stored in the nvs namespace will also be erased besides been cleaned from the mesh stack.

b. If Provisioner tries to work properly again, we can invoke the open function to open a new nvs namespace or a previously added one, and restore the mesh information from it if not erased.
c. The working process shall be as following: a) Open settings A b) Start to provision and control nodes c) Close settings A d) Open settings B e) Start to provision and control other nodes f) Close settings B g) ……

Parameters
- **index** –**[in]** Settings index.
- **erase** –**[in]** Indicate if erasing mesh information.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_close_settings_with_uid**(const char *uid, bool erase)

This function is called by Provisioner to close a nvs namespace which is opened previously for storing mesh information.

Note: 1. Before closing the nvs namespace, it must be open.

a. When the function is invoked, the Provisioner functionality will be disabled firstly, and: a) If the "erase" flag is set to false, the mesh information will be cleaned (e.g. removing NetKey, AppKey, nodes, etc) from the mesh stack. b) If the "erase" flag is set to true, the mesh information stored in the nvs namespace will also be erased besides been cleaned from the mesh stack.
b. If Provisioner tries to work properly again, we can invoke the open function to open a new nvs namespace or a previously added one, and restore the mesh information from it if not erased.
c. The working process shall be as following: a) Open settings A b) Start to provision and control nodes c) Close settings A d) Open settings B e) Start to provision and control other nodes f) Close settings B g) ……

Parameters
- **uid** –**[in]** Settings user id.
- **erase** –**[in]** Indicate if erasing mesh information.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_delete_settings_with_index**(uint8_t index)

This function is called by Provisioner to erase the mesh information and settings user id from a nvs namespace.

Note: When this function is called, the nvs namespace must not be open. This function is used to erase the mesh information and settings user id which are not used currently.

Parameters **index** –**[in]** Settings index.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_delete_settings_with_uid**(const char *uid)

This function is called by Provisioner to erase the mesh information and settings user id from a nvs namespace.

Note: When this function is called, the nvs namespace must not be open. This function is used to erase the mesh information and settings user id which are not used currently.

Parameters **uid** –**[in]** Settings user id.
Returns ESP_OK on success or error code otherwise.

const char *`esp_ble_mesh_provisioner_get_settings_uid`(uint8_t index)

This function is called by Provisioner to get settings user id.

Parameters `index` –**[in]** Settings index.
Returns Setting user id on success or NULL on failure.

uint8_t `esp_ble_mesh_provisioner_get_settings_index`(const char *uid)

This function is called by Provisioner to get settings index.

Parameters `uid` –**[in]** Settings user id.
Returns Settings index.

uint8_t `esp_ble_mesh_provisioner_get_free_settings_count`(void)

This function is called by Provisioner to get the number of free settings user id.

Returns Number of free settings user id.

const uint8_t *`esp_ble_mesh_get_fast_prov_app_key`(uint16_t net_idx, uint16_t app_idx)

This function is called to get fast provisioning application key.

Parameters
- `net_idx` –**[in]** Network key index.
- `app_idx` –**[in]** Application key index.

Returns Application key on success, or NULL on failure.

Type Definitions

typedef void (*`esp_ble_mesh_model_cb_t`)(*esp_ble_mesh_model_cb_event_t* event, *esp_ble_mesh_model_cb_param_t* *param)

: event, event code of user-defined model events; param, parameters of user-defined model events

ESP-BLE-MESH Node/Provisioner Provisioning

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_provisioning_api.h

Functions

esp_err_t `esp_ble_mesh_register_prov_callback`(*esp_ble_mesh_prov_cb_t* callback)

Register BLE Mesh provisioning callback.

Parameters `callback` –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

bool `esp_ble_mesh_node_is_provisioned`(void)

Check if a device has been provisioned.

Returns TRUE if the device is provisioned, FALSE if the device is unprovisioned.

esp_err_t `esp_ble_mesh_node_prov_enable`(*esp_ble_mesh_prov_bearer_t* bearers)

Enable specific provisioning bearers to get the device ready for provisioning.

Note: PB-ADV: send unprovisioned device beacon. PB-GATT: send connectable advertising packets.

Parameters `bearers` –Bit-wise OR of provisioning bearers.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_prov_disable**(*esp_ble_mesh_prov_bearer_t* bearers)

Disable specific provisioning bearers to make a device inaccessible for provisioning.

Parameters **bearers** –Bit-wise OR of provisioning bearers.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_set_oob_pub_key**(uint8_t pub_key_x[32], uint8_t pub_key_y[32], uint8_t private_key[32])

Unprovisioned device set own oob public key & private key pair.

Note: In order to avoid suffering brute-forcing attack (CVE-2020-26559). The Bluetooth SIG recommends that potentially vulnerable mesh provisioners use an out-of-band mechanism to exchange the public keys. So as an unprovisioned device, it should use this function to input the Public Key exchanged through the out-of-band mechanism.

Parameters

- **pub_key_x** –**[in]** Unprovisioned device' s Public Key X
- **pub_key_y** –**[in]** Unprovisioned device' s Public Key Y
- **private_key** –**[in]** Unprovisioned device' s Private Key

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_input_number**(uint32_t number)

Provide provisioning input OOB number.

Note: This is intended to be called if the user has received ESP_BLE_MESH_NODE_PROV_INPUT_EVT with ESP_BLE_MESH_ENTER_NUMBER as the action.

Parameters **number** –**[in]** Number input by device.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_node_input_string**(const char *string)

Provide provisioning input OOB string.

Note: This is intended to be called if the user has received ESP_BLE_MESH_NODE_PROV_INPUT_EVT with ESP_BLE_MESH_ENTER_STRING as the action.

Parameters **string** –**[in]** String input by device.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_set_unprovisioned_device_name**(const char *name)

Using this function, an unprovisioned device can set its own device name, which will be broadcasted in its advertising data.

Note: This API applicable to PB-GATT mode only by setting the name to the scan response data, it doesn' t apply to PB-ADV mode.

Parameters **name** –**[in]** Unprovisioned device name
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_read_oob_pub_key**(uint8_t link_idx, uint8_t pub_key_x[32], uint8_t pub_key_y[32])

Provisioner inputs unprovisioned device's oob public key.

Note: In order to avoid suffering brute-forcing attack (CVE-2020-26559). The Bluetooth SIG recommends that potentially vulnerable mesh provisioners use an out-of-band mechanism to exchange the public keys.

Parameters
- **link_idx** –**[in]** The provisioning link index
- **pub_key_x** –**[in]** Unprovisioned device's Public Key X
- **pub_key_y** –**[in]** Unprovisioned device's Public Key Y

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_input_string**(const char *string, uint8_t link_idx)

Provide provisioning input OOB string.

```
        This is intended to be called after the esp_ble_mesh_prov_t prov_
↪input_num
        callback has been called with ESP_BLE_MESH_ENTER_STRING as the␣
↪action.
```

Parameters
- **string** –**[in]** String input by Provisioner.
- **link_idx** –**[in]** The provisioning link index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_input_number**(uint32_t number, uint8_t link_idx)

Provide provisioning input OOB number.

```
        This is intended to be called after the esp_ble_mesh_prov_t prov_
↪input_num
        callback has been called with ESP_BLE_MESH_ENTER_NUMBER as the␣
↪action.
```

Parameters
- **number** –**[in]** Number input by Provisioner.
- **link_idx** –**[in]** The provisioning link index.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_prov_enable**(*esp_ble_mesh_prov_bearer_t* bearers)

Enable one or more provisioning bearers.

Note: PB-ADV: Enable BLE scan. PB-GATT: Initialize corresponding BLE Mesh Proxy info.

Parameters **bearers** –**[in]** Bit-wise OR of provisioning bearers.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_prov_disable**(*esp_ble_mesh_prov_bearer_t* bearers)

Disable one or more provisioning bearers.

Note: PB-ADV: Disable BLE scan. PB-GATT: Break any existing BLE Mesh Provisioning connections.

Parameters **bearers** –**[in]** Bit-wise OR of provisioning bearers.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_add_unprov_dev**(*esp_ble_mesh_unprov_dev_add_t* *add_dev, *esp_ble_mesh_dev_add_flag_t* flags)

Add unprovisioned device info to the unprov_dev queue.

Note: : 1. Currently address type only supports public address and static random address.

a. If device UUID and/or device address as well as address type already exist in the device queue, but the bearer is different from the existing one, add operation will also be successful and it will update the provision bearer supported by the device.
b. For example, if the Provisioner wants to add an unprovisioned device info before receiving its unprovisioned device beacon or Mesh Provisioning advertising packets, the Provisioner can use this API to add the device info with each one or both of device UUID and device address added. When the Provisioner gets the device's advertising packets, it will start provisioning the device internally.
   - In this situation, the Provisioner can set bearers with each one or both of ESP_BLE_MESH_PROV_ADV and ESP_BLE_MESH_PROV_GATT enabled, and cannot set flags with ADD_DEV_START_PROV_NOW_FLAG enabled.
c. Another example is when the Provisioner receives the unprovisioned device's beacon or Mesh Provisioning advertising packets, the advertising packets will be reported on to the application layer using the callback registered by the function esp_ble_mesh_register_prov_callback. And in the callback, the Provisioner can call this API to start provisioning the device.
   - If the Provisioner uses PB-ADV to provision, either one or both of device UUID and device address can be added, bearers shall be set with ESP_BLE_MESH_PROV_ADV enabled and the flags shall be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
   - If the Provisioner uses PB-GATT to provision, both the device UUID and device address need to be added, bearers shall be set with ESP_BLE_MESH_PROV_GATT enabled, and the flags shall be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
   - If the Provisioner just wants to store the unprovisioned device info when receiving its advertising packets and start to provision it the next time (e.g. after receiving its advertising packets again), then it can add the device info with either one or both of device UUID and device address included. Bearers can be set with either one or both of ESP_BLE_MESH_PROV_ADV and ESP_BLE_MESH_PROV_GATT enabled (recommend to enable the bearer which will receive its advertising packets, because if the other bearer is enabled, the Provisioner is not aware if the device supports the bearer), and flags cannot be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
   - Note: ESP_BLE_MESH_PROV_ADV, ESP_BLE_MESH_PROV_GATT and ADD_DEV_START_PROV_NOW_FLAG can not be enabled at the same time.

Parameters
- **add_dev** –**[in]** Pointer to a struct containing the device information
- **flags** –**[in]** Flags indicate several operations on the device information
  – Remove device information from queue after device has been provisioned (BIT0)
  – Start provisioning immediately after device is added to queue (BIT1)
  – Device can be removed if device queue is full (BIT2)

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_prov_device_with_addr**(const uint8_t uuid[16], *esp_ble_mesh_bd_addr_t* addr, *esp_ble_mesh_addr_type_t* addr_type, *esp_ble_mesh_prov_bearer_t* bearer, uint16_t oob_info, uint16_t unicast_addr)

Provision an unprovisioned device and assign a fixed unicast address for it in advance.

Note: : 1. Currently address type only supports public address and static random address.

a. Bearer must be equal to ESP_BLE_MESH_PROV_ADV or ESP_BLE_MESH_PROV_GATT, since Provisioner will start to provision a device immediately once this function is invoked. And the input bearer must be identical with the one within the parameters of the ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT event.
b. If this function is used by a Provisioner to provision devices, the application should take care of the assigned unicast address and avoid overlap of the unicast addresses of different nodes.
c. Recommend to use only one of the functions "esp_ble_mesh_provisioner_add_unprov_dev" and "esp_ble_mesh_provisioner_prov_device_with_addr" by a Provisioner.

Parameters

- **uuid** –**[in]** Device UUID of the unprovisioned device
- **addr** –**[in]** Device address of the unprovisioned device
- **addr_type** –**[in]** Device address type of the unprovisioned device
- **bearer** –**[in]** Provisioning bearer going to be used by Provisioner
- **oob_info** –**[in]** OOB info of the unprovisioned device
- **unicast_addr** –**[in]** Unicast address going to be allocated for the unprovisioned device

Returns Zero on success or (negative) error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_delete_dev**(*esp_ble_mesh_device_delete_t* *del_dev)

Delete device from queue, and reset current provisioning link with the device.

Note: If the device is in the queue, remove it from the queue; if the device is being provisioned, terminate the provisioning procedure. Either one of the device address or device UUID can be used as input.

Parameters **del_dev** –**[in]** Pointer to a struct containing the device information.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_dev_uuid_match**(const uint8_t *match_val, uint8_t match_len, uint8_t offset, bool prov_after_match)

This function is called by Provisioner to set the part of the device UUID to be compared before starting to provision.

Parameters

- **match_val** –**[in]** Value to be compared with the part of the device UUID.
- **match_len** –**[in]** Length of the compared match value.
- **offset** –**[in]** Offset of the device UUID to be compared (based on zero).
- **prov_after_match** –**[in]** Flag used to indicate whether provisioner should start to provision the device immediately if the part of the UUID matches.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_prov_data_info**(*esp_ble_mesh_prov_data_info_t* *prov_data_info)

This function is called by Provisioner to set provisioning data information before starting to provision.

Parameters **prov_data_info** –**[in]** Pointer to a struct containing net_idx or flags or iv_index.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_static_oob_value**(const uint8_t *value, uint8_t length)

This function is called by Provisioner to set static oob value used for provisioning.

AuthValues selected using a cryptographically secure random or pseudorandom number generator and having the maximum permitted entropy (128-bits) will be most difficult to brute-force. AuthValues with reduced entropy or generated in a predictable manner will not grant the same level of protection against this vulnerability. Selecting a new AuthValue with each provisioning attempt can also make it more difficult to launch a brute-force attack by requiring the attacker to restart the search with each provisioning attempt (CVE-2020-26556).

Note: The Bluetooth SIG recommends that mesh implementations enforce a randomly selected AuthValue using all of the available bits, where permitted by the implementation. A large entropy helps ensure that a brute-force of the AuthValue, even a static AuthValue, cannot normally be completed in a reasonable time (CVE-2020-26557).

Parameters
- **value** –**[in]** Pointer to the static oob value.
- **length** –**[in]** Length of the static oob value.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_primary_elem_addr**(uint16_t addr)

This function is called by Provisioner to set own Primary element address.

Note: This API must be invoked when BLE Mesh initialization is completed successfully, and can be invoked before Provisioner functionality is enabled. Once this API is invoked successfully, the prov_unicast_addr value in the struct *esp_ble_mesh_prov_t* will be ignored, and Provisioner will use this address as its own primary element address. And if the unicast address going to assigned for the next unprovisioned device is smaller than the input address + element number of Provisioner, then the address for the next unprovisioned device will be recalculated internally.

Parameters **addr** –**[in]** Unicast address of the Primary element of Provisioner.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_set_fast_prov_info**(*esp_ble_mesh_fast_prov_info_t* *fast_prov_info)

This function is called to set provisioning data information before starting fast provisioning.

Parameters **fast_prov_info** –**[in]** Pointer to a struct containing unicast address range, net_idx, etc.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_set_fast_prov_action**(*esp_ble_mesh_fast_prov_action_t* action)

This function is called to start/suspend/exit fast provisioning.

Parameters **action** –**[in]** fast provisioning action (i.e. enter, suspend, exit).
Returns ESP_OK on success or error code otherwise.

Type Definitions

typedef void (***esp_ble_mesh_prov_cb_t**)(*esp_ble_mesh_prov_cb_event_t* event, *esp_ble_mesh_prov_cb_param_t* *param)

: event, event code of provisioning events; param, parameters of provisioning events

typedef void (*`esp_ble_mesh_prov_adv_cb_t`)(const *esp_ble_mesh_bd_addr_t* addr, const *esp_ble_mesh_addr_type_t* addr_type, const uint8_t adv_type, const uint8_t *dev_uuid, uint16_t oob_info, *esp_ble_mesh_prov_bearer_t* bearer)

Callback for Provisioner that received advertising packets from unprovisioned devices which are not in the unprovisioned device queue.

Report on the unprovisioned device beacon and mesh provisioning service adv data to application.

Param addr **[in]** Pointer to the unprovisioned device address.
Param addr_type **[in]** Unprovisioned device address type.
Param adv_type **[in]** Adv packet type(ADV_IND or ADV_NONCONN_IND).
Param dev_uuid **[in]** Unprovisioned device UUID pointer.
Param oob_info **[in]** OOB information of the unprovisioned device.
Param bearer **[in]** Adv packet received from PB-GATT or PB-ADV bearer.

ESP-BLE-MESH GATT Proxy Server

Header File

- components/bt/esp_ble_mesh/api/core/include/esp_ble_mesh_proxy_api.h

Functions

esp_err_t `esp_ble_mesh_proxy_identity_enable`(void)

Enable advertising with Node Identity.

Note: This API requires that GATT Proxy support be enabled. Once called, each subnet starts advertising using Node Identity for the next 60 seconds, and after 60s Network ID will be advertised. Under normal conditions, the BLE Mesh Proxy Node Identity and Network ID advertising will be enabled automatically by BLE Mesh stack after the device is provisioned.

Returns ESP_OK on success or error code otherwise.

esp_err_t `esp_ble_mesh_proxy_gatt_enable`(void)

Enable BLE Mesh GATT Proxy Service.

Returns ESP_OK on success or error code otherwise.

esp_err_t `esp_ble_mesh_proxy_gatt_disable`(void)

Disconnect the BLE Mesh GATT Proxy connection if there is any, and disable the BLE Mesh GATT Proxy Service.

Returns ESP_OK on success or error code otherwise.

esp_err_t `esp_ble_mesh_proxy_client_connect`(*esp_ble_mesh_bd_addr_t* addr, *esp_ble_mesh_addr_type_t* addr_type, uint16_t net_idx)

Proxy Client creates a connection with the Proxy Server.

Parameters
- `addr` –**[in]** Device address of the Proxy Server.
- `addr_type` –**[in]** Device address type(public or static random).
- `net_idx` –**[in]** NetKey Index related with Network ID in the Mesh Proxy advertising packet.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_client_disconnect**(uint8_t conn_handle)

Proxy Client terminates a connection with the Proxy Server.

Parameters **conn_handle** –**[in]** Proxy connection handle.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_client_set_filter_type**(uint8_t conn_handle, uint16_t net_idx, *esp_ble_mesh_proxy_filter_type_t* filter_type)

Proxy Client sets the filter type of the Proxy Server.

Parameters
- **conn_handle** –**[in]** Proxy connection handle.
- **net_idx** –**[in]** Corresponding NetKey Index.
- **filter_type** –**[in]** whitelist or blacklist.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_client_add_filter_addr**(uint8_t conn_handle, uint16_t net_idx, uint16_t *addr, uint16_t addr_num)

Proxy Client adds address to the Proxy Server filter list.

Parameters
- **conn_handle** –**[in]** Proxy connection handle.
- **net_idx** –**[in]** Corresponding NetKey Index.
- **addr** –**[in]** Pointer to the filter address.
- **addr_num** –**[in]** Number of the filter address.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_client_remove_filter_addr**(uint8_t conn_handle, uint16_t net_idx, uint16_t *addr, uint16_t addr_num)

Proxy Client removes address from the Proxy Server filter list.

Parameters
- **conn_handle** –**[in]** Proxy connection handle.
- **net_idx** –**[in]** Corresponding NetKey Index.
- **addr** –**[in]** Pointer to the filter address.
- **addr_num** –**[in]** Number of the filter address.

Returns ESP_OK on success or error code otherwise.

ESP-BLE-MESH Models API Reference

This section contains ESP-BLE-MESH Model related APIs, event types, event parameters, etc.

There are six categories of models:

- *Configuration Client/Server Models*
- *Health Client/Server Models*
- *Generic Client/Server Models*
- *Sensor Client/Server Models*
- *Time and Scenes Client/Server Models*
- *Lighting Client/Server Models*

Note: Definitions related to Server Models are being updated, and will be released soon.

Configuration Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_config_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_config_client_callback**(*esp_ble_mesh_cfg_client_cb_t* callback)

Register BLE Mesh Config Client Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_config_server_callback**(*esp_ble_mesh_cfg_server_cb_t* callback)

Register BLE Mesh Config Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_config_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_cfg_client_get_state_t* *get_state)

Get the value of Config Server Model states using the Config Client Model get messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_opcode_config_client_get_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_config_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_cfg_client_set_state_t* *set_state)

Set the value of the Configuration Server Model states using the Config Client Model set messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_opcode_config_client_set_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_cfg_client_get_state_t**

#include <esp_ble_mesh_config_model_api.h> For ESP_BLE_MESH_MODEL_OP_BEACON_GET ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET ESP_BLE_MESH_MODEL_OP_RELAY_GET ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET ESP_BLE_MESH_MODEL_OP_FRIEND_GET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_GET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_GET the get_state parameter in the esp_ble_mesh_config_client_get_state function should not be set to NULL.

Public Members

esp_ble_mesh_cfg_model_pub_get_t **model_pub_get**

For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET.

esp_ble_mesh_cfg_composition_data_get_t **comp_data_get**

For ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET.

esp_ble_mesh_cfg_sig_model_sub_get_t **sig_model_sub_get**

For ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_GET

esp_ble_mesh_cfg_vnd_model_sub_get_t **vnd_model_sub_get**

For ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET

esp_ble_mesh_cfg_app_key_get_t **app_key_get**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_GET.

esp_ble_mesh_cfg_node_identity_get_t **node_identity_get**

For ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET.

esp_ble_mesh_cfg_sig_model_app_get_t **sig_model_app_get**

For ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_GET

esp_ble_mesh_cfg_vnd_model_app_get_t **vnd_model_app_get**

For ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET

esp_ble_mesh_cfg_kr_phase_get_t **kr_phase_get**

For ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_GET

esp_ble_mesh_cfg_lpn_polltimeout_get_t **lpn_pollto_get**

For ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_GET

union **esp_ble_mesh_cfg_client_set_state_t**

#include <*esp_ble_mesh_config_model_api.h*> For ESP_BLE_MESH_MODEL_OP_BEACON_SET ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET ESP_BLE_MESH_MODEL_OP_RELAY_SET ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_AD ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUA ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND ESP_BLE_MESH_MODEL_OP_NODE_RESET ESP_BLE_MESH_MODEL_OP_FRIEND_SET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET the set_state parameter in the esp_ble_mesh_config_client_set_state function should not be set to NULL.

Public Members

esp_ble_mesh_cfg_beacon_set_t **beacon_set**

For ESP_BLE_MESH_MODEL_OP_BEACON_SET

esp_ble_mesh_cfg_default_ttl_set_t **default_ttl_set**

For ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET

esp_ble_mesh_cfg_friend_set_t **friend_set**

For ESP_BLE_MESH_MODEL_OP_FRIEND_SET

esp_ble_mesh_cfg_gatt_proxy_set_t **gatt_proxy_set**

For ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET

esp_ble_mesh_cfg_relay_set_t **relay_set**

For ESP_BLE_MESH_MODEL_OP_RELAY_SET

esp_ble_mesh_cfg_net_key_add_t **net_key_add**

For ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD

esp_ble_mesh_cfg_app_key_add_t **app_key_add**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD

esp_ble_mesh_cfg_model_app_bind_t **model_app_bind**

For ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND

esp_ble_mesh_cfg_model_pub_set_t **model_pub_set**

For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET

esp_ble_mesh_cfg_model_sub_add_t **model_sub_add**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD

esp_ble_mesh_cfg_model_sub_delete_t **model_sub_delete**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE

esp_ble_mesh_cfg_model_sub_overwrite_t **model_sub_overwrite**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE

esp_ble_mesh_cfg_model_sub_va_add_t **model_sub_va_add**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD

esp_ble_mesh_cfg_model_sub_va_delete_t **model_sub_va_delete**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELETE

esp_ble_mesh_cfg_model_sub_va_overwrite_t **model_sub_va_overwrite**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVERWRITE

esp_ble_mesh_cfg_heartbeat_pub_set_t **heartbeat_pub_set**

For ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET

esp_ble_mesh_cfg_heartbeat_sub_set_t **heartbeat_sub_set**

For ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET

esp_ble_mesh_cfg_model_pub_va_set_t **model_pub_va_set**

For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET

esp_ble_mesh_cfg_model_sub_delete_all_t **model_sub_delete_all**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL

esp_ble_mesh_cfg_net_key_update_t **net_key_update**

For ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE

esp_ble_mesh_cfg_net_key_delete_t **net_key_delete**

For ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE

esp_ble_mesh_cfg_app_key_update_t **app_key_update**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE

esp_ble_mesh_cfg_app_key_delete_t **app_key_delete**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE

esp_ble_mesh_cfg_node_identity_set_t **node_identity_set**

For ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET

esp_ble_mesh_cfg_model_app_unbind_t **model_app_unbind**

For ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND

esp_ble_mesh_cfg_kr_phase_set_t **kr_phase_set**

For ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_SET

esp_ble_mesh_cfg_net_transmit_set_t **net_transmit_set**

For ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET

union **esp_ble_mesh_cfg_client_common_cb_param_t**

#include <esp_ble_mesh_config_model_api.h> Configuration Client Model received message union.

Public Members

esp_ble_mesh_cfg_beacon_status_cb_t **beacon_status**

The beacon status value

esp_ble_mesh_cfg_comp_data_status_cb_t **comp_data_status**

The composition data status value

esp_ble_mesh_cfg_default_ttl_status_cb_t **default_ttl_status**

The default_ttl status value

esp_ble_mesh_cfg_gatt_proxy_status_cb_t **gatt_proxy_status**

The gatt_proxy status value

esp_ble_mesh_cfg_relay_status_cb_t **relay_status**

The relay status value

esp_ble_mesh_cfg_model_pub_status_cb_t **model_pub_status**

The model publication status value

esp_ble_mesh_cfg_model_sub_status_cb_t **model_sub_status**

The model subscription status value

esp_ble_mesh_cfg_net_key_status_cb_t **netkey_status**

The netkey status value

esp_ble_mesh_cfg_app_key_status_cb_t **appkey_status**

The appkey status value

esp_ble_mesh_cfg_mod_app_status_cb_t **model_app_status**

The model app status value

esp_ble_mesh_cfg_friend_status_cb_t **friend_status**

The friend status value

esp_ble_mesh_cfg_hb_pub_status_cb_t **heartbeat_pub_status**

The heartbeat publication status value

esp_ble_mesh_cfg_hb_sub_status_cb_t **heartbeat_sub_status**

The heartbeat subscription status value

esp_ble_mesh_cfg_net_trans_status_cb_t **net_transmit_status**

The network transmit status value

esp_ble_mesh_cfg_model_sub_list_cb_t **model_sub_list**

The model subscription list value

esp_ble_mesh_cfg_net_key_list_cb_t **netkey_list**

The network key index list value

esp_ble_mesh_cfg_app_key_list_cb_t **appkey_list**

The application key index list value

esp_ble_mesh_cfg_node_id_status_cb_t **node_identity_status**

The node identity status value

esp_ble_mesh_cfg_model_app_list_cb_t **model_app_list**

The model application key index list value

esp_ble_mesh_cfg_kr_phase_status_cb_t **kr_phase_status**

The key refresh phase status value

esp_ble_mesh_cfg_lpn_pollto_status_cb_t **lpn_timeout_status**

The low power node poll timeout status value

union **esp_ble_mesh_cfg_server_state_change_t**

#include <esp_ble_mesh_config_model_api.h> Configuration Server model state change value union.

Public Members

esp_ble_mesh_state_change_cfg_mod_pub_set_t **mod_pub_set**

The recv_op in ctx can be used to decide which state is changed. Config Model Publication Set

esp_ble_mesh_state_change_cfg_model_sub_add_t **mod_sub_add**

Config Model Subscription Add

esp_ble_mesh_state_change_cfg_model_sub_delete_t **mod_sub_delete**

Config Model Subscription Delete

esp_ble_mesh_state_change_cfg_netkey_add_t **netkey_add**

Config NetKey Add

esp_ble_mesh_state_change_cfg_netkey_update_t **netkey_update**

Config NetKey Update

esp_ble_mesh_state_change_cfg_netkey_delete_t **netkey_delete**

Config NetKey Delete

esp_ble_mesh_state_change_cfg_appkey_add_t **appkey_add**

Config AppKey Add

esp_ble_mesh_state_change_cfg_appkey_update_t **appkey_update**

Config AppKey Update

esp_ble_mesh_state_change_cfg_appkey_delete_t **appkey_delete**

Config AppKey Delete

esp_ble_mesh_state_change_cfg_model_app_bind_t **mod_app_bind**

Config Model App Bind

esp_ble_mesh_state_change_cfg_model_app_unbind_t **mod_app_unbind**

Config Model App Unbind

esp_ble_mesh_state_change_cfg_kr_phase_set_t **kr_phase_set**

Config Key Refresh Phase Set

union **esp_ble_mesh_cfg_server_cb_value_t**

#include <esp_ble_mesh_config_model_api.h> Configuration Server model callback value union.

Public Members

esp_ble_mesh_cfg_server_state_change_t **state_change**

ESP_BLE_MESH_CFG_SERVER_STATE_CHANGE_EVT

Structures

struct **esp_ble_mesh_cfg_srv**

Configuration Server Model context

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Configuration Server Model

uint8_t **net_transmit**

Network Transmit state

uint8_t **relay**

Relay Mode state

uint8_t **relay_retransmit**

Relay Retransmit state

uint8_t **beacon**

Secure Network Beacon state

uint8_t **gatt_proxy**

GATT Proxy state

uint8_t **friend_state**

Friend state

uint8_t **default_ttl**

Default TTL

struct k_delayed_work **timer**

Heartbeat Publication timer

uint16_t **dst**

Destination address for Heartbeat messages

uint16_t **count**

Number of Heartbeat messages to be sent

Number of Heartbeat messages received

uint8_t **period**

Period for sending Heartbeat messages

uint8_t **ttl**

TTL to be used when sending Heartbeat messages

uint16_t **feature**

Bit field indicating features that trigger Heartbeat messages when changed

uint16_t **net_idx**

NetKey Index used by Heartbeat Publication

struct *esp_ble_mesh_cfg_srv*::[anonymous] **heartbeat_pub**

Heartbeat Publication

int64_t **expiry**

Timestamp when Heartbeat subscription period is expired

uint16_t **src**

Source address for Heartbeat messages

uint8_t **min_hops**

Minimum hops when receiving Heartbeat messages

uint8_t **max_hops**

Maximum hops when receiving Heartbeat messages

esp_ble_mesh_cb_t **heartbeat_recv_cb**

Optional heartbeat subscription tracking function

struct *esp_ble_mesh_cfg_srv*::[anonymous] **heartbeat_sub**

Heartbeat Subscription

struct **esp_ble_mesh_cfg_composition_data_get_t**

Parameters of Config Composition Data Get.

Public Members

uint8_t **page**

Page number of the Composition Data.

struct **esp_ble_mesh_cfg_model_pub_get_t**

Parameters of Config Model Publication Get.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_sig_model_sub_get_t**

Parameters of Config SIG Model Subscription Get.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

struct **esp_ble_mesh_cfg_vnd_model_sub_get_t**

Parameters of Config Vendor Model Subscription Get.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_app_key_get_t**

Parameters of Config AppKey Get.

Public Members

uint16_t **net_idx**

The network key index

struct **esp_ble_mesh_cfg_node_identity_get_t**

Parameters of Config Node Identity Get.

Public Members

uint16_t **net_idx**

The network key index

struct **esp_ble_mesh_cfg_sig_model_app_get_t**

Parameters of Config SIG Model App Get.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

struct **esp_ble_mesh_cfg_vnd_model_app_get_t**

Parameters of Config Vendor Model App Get.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_kr_phase_get_t**

Parameters of Config Key Refresh Phase Get.

Public Members

uint16_t **net_idx**

The network key index

struct **esp_ble_mesh_cfg_lpn_polltimeout_get_t**

Parameters of Config Low Power Node PollTimeout Get.

Public Members

uint16_t **lpn_addr**

The unicast address of the Low Power node

struct **esp_ble_mesh_cfg_beacon_set_t**

Parameters of Config Beacon Set.

Public Members

uint8_t **beacon**

New Secure Network Beacon state

struct **esp_ble_mesh_cfg_default_ttl_set_t**

Parameters of Config Default TTL Set.

Public Members

uint8_t **ttl**

The default TTL state value

struct **esp_ble_mesh_cfg_friend_set_t**

Parameters of Config Friend Set.

Public Members

uint8_t **friend_state**

The friend state value

struct **esp_ble_mesh_cfg_gatt_proxy_set_t**

Parameters of Config GATT Proxy Set.

Public Members

uint8_t **gatt_proxy**

The GATT Proxy state value

struct **esp_ble_mesh_cfg_relay_set_t**

Parameters of Config Relay Set.

Public Members

uint8_t **relay**

The relay value

uint8_t **relay_retransmit**

The relay retransmit value

struct **esp_ble_mesh_cfg_net_key_add_t**

Parameters of Config NetKey Add.

Public Members

uint16_t **net_idx**

The network key index

uint8_t **net_key**[16]

The network key value

struct **esp_ble_mesh_cfg_app_key_add_t**

Parameters of Config AppKey Add.

Public Members

uint16_t **net_idx**

The network key index

uint16_t **app_idx**

The app key index

uint8_t **app_key**[16]

The app key value

struct **esp_ble_mesh_cfg_model_app_bind_t**

Parameters of Config Model App Bind.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_app_idx**

Index of the app key to bind with the model

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_pub_set_t**

Parameters of Config Model Publication Set.

Public Members

uint16_t **element_addr**

The element address

uint16_t **publish_addr**

Value of the publish address

uint16_t **publish_app_idx**

Index of the application key

bool **cred_flag**

Value of the Friendship Credential Flag

uint8_t **publish_ttl**

Default TTL value for the publishing messages

uint8_t **publish_period**

Period for periodic status publishing

uint8_t **publish_retransmit**

Number of retransmissions and number of 50-millisecond steps between retransmissions

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_add_t**

Parameters of Config Model Subscription Add.

Public Members

uint16_t **element_addr**

The element address

uint16_t **sub_addr**

The address to be added to the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_delete_t**

Parameters of Config Model Subscription Delete.

Public Members

uint16_t **element_addr**

The element address

uint16_t **sub_addr**

The address to be removed from the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_overwrite_t**

Parameters of Config Model Subscription Overwrite.

Public Members

uint16_t **element_addr**

The element address

uint16_t **sub_addr**

The address to be added to the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_va_add_t**

Parameters of Config Model Subscription Virtual Address Add.

Public Members

uint16_t **element_addr**

The element address

uint8_t **label_uuid**[16]

The Label UUID of the virtual address to be added to the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_va_delete_t**

Parameters of Config Model Subscription Virtual Address Delete.

Public Members

uint16_t **element_addr**

The element address

uint8_t **label_uuid**[16]

The Label UUID of the virtual address to be removed from the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_va_overwrite_t**

Parameters of Config Model Subscription Virtual Address Overwrite.

Public Members

uint16_t **element_addr**

The element address

uint8_t **label_uuid**[16]

The Label UUID of the virtual address to be added to the Subscription List

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_pub_va_set_t**

Parameters of Config Model Publication Virtual Address Set.

Public Members

uint16_t **element_addr**

The element address

uint8_t **label_uuid**[16]

Value of the Label UUID publish address

uint16_t **publish_app_idx**

Index of the application key

bool **cred_flag**

Value of the Friendship Credential Flag

uint8_t **publish_ttl**

Default TTL value for the publishing messages

uint8_t **publish_period**

Period for periodic status publishing

uint8_t **publish_retransmit**

Number of retransmissions and number of 50-millisecond steps between retransmissions

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_model_sub_delete_all_t**

Parameters of Config Model Subscription Delete All.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_net_key_update_t**

Parameters of Config NetKey Update.

Public Members

uint16_t **net_idx**

The network key index

uint8_t **net_key**[16]

The network key value

struct **esp_ble_mesh_cfg_net_key_delete_t**

Parameters of Config NetKey Delete.

Public Members

uint16_t **net_idx**

The network key index

struct **esp_ble_mesh_cfg_app_key_update_t**

Parameters of Config AppKey Update.

Public Members

uint16_t **net_idx**

The network key index

uint16_t **app_idx**

The app key index

uint8_t **app_key**[16]

The app key value

struct **esp_ble_mesh_cfg_app_key_delete_t**

Parameters of Config AppKey Delete.

Public Members

uint16_t **net_idx**

The network key index

uint16_t **app_idx**

The app key index

struct **esp_ble_mesh_cfg_node_identity_set_t**

Parameters of Config Node Identity Set.

Public Members

uint16_t **net_idx**

The network key index

uint8_t **identity**

New Node Identity state

struct **esp_ble_mesh_cfg_model_app_unbind_t**

Parameters of Config Model App Unbind.

Public Members

uint16_t **element_addr**

The element address

uint16_t **model_app_idx**

Index of the app key to bind with the model

uint16_t **model_id**

The model id

uint16_t **company_id**

The company id, if not a vendor model, shall set to 0xFFFF

struct **esp_ble_mesh_cfg_kr_phase_set_t**

Parameters of Config Key Refresh Phase Set.

Public Members

uint16_t **net_idx**

The network key index

uint8_t **transition**

New Key Refresh Phase Transition

struct **esp_ble_mesh_cfg_net_transmit_set_t**

Parameters of Config Network Transmit Set.

Public Members

uint8_t **net_transmit**

Network Transmit State

struct **esp_ble_mesh_cfg_heartbeat_pub_set_t**

Parameters of Config Model Heartbeat Publication Set.

Public Members

uint16_t **dst**

Destination address for Heartbeat messages

uint8_t **count**

Number of Heartbeat messages to be sent

uint8_t **period**

Period for sending Heartbeat messages

uint8_t **ttl**

TTL to be used when sending Heartbeat messages

uint16_t **feature**

Bit field indicating features that trigger Heartbeat messages when changed

uint16_t **net_idx**

NetKey Index

struct **esp_ble_mesh_cfg_heartbeat_sub_set_t**

Parameters of Config Model Heartbeat Subscription Set.

Public Members

uint16_t **src**

Source address for Heartbeat messages

uint16_t **dst**

Destination address for Heartbeat messages

uint8_t **period**

Period for receiving Heartbeat messages

struct **esp_ble_mesh_cfg_beacon_status_cb_t**

Parameter of Config Beacon Status

Public Members

uint8_t **beacon**

Secure Network Beacon state value

struct **esp_ble_mesh_cfg_comp_data_status_cb_t**

Parameters of Config Composition Data Status

Public Members

uint8_t **page**

Page number of the Composition Data

struct net_buf_simple ***composition_data**

Pointer to Composition Data for the identified page

struct **esp_ble_mesh_cfg_default_ttl_status_cb_t**

Parameter of Config Default TTL Status

Public Members

uint8_t **default_ttl**

Default TTL state value

struct **esp_ble_mesh_cfg_gatt_proxy_status_cb_t**

Parameter of Config GATT Proxy Status

Public Members

uint8_t **gatt_proxy**

GATT Proxy state value

struct **esp_ble_mesh_cfg_relay_status_cb_t**

Parameters of Config Relay Status

Public Members

uint8_t **relay**

Relay state value

uint8_t **retransmit**

Relay retransmit value(number of retransmissions and number of 10-millisecond steps between retransmissions)

struct **esp_ble_mesh_cfg_model_pub_status_cb_t**

Parameters of Config Model Publication Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **element_addr**

Address of the element

uint16_t **publish_addr**

Value of the publish address

uint16_t **app_idx**

Index of the application key

bool **cred_flag**

Value of the Friendship Credential Flag

uint8_t **ttl**

Default TTL value for the outgoing messages

uint8_t **period**

Period for periodic status publishing

uint8_t **transmit**

Number of retransmissions and number of 50-millisecond steps between retransmissions

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_cfg_model_sub_status_cb_t**

Parameters of Config Model Subscription Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **element_addr**

Address of the element

uint16_t **sub_addr**

Value of the address

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_cfg_net_key_status_cb_t**

Parameters of Config NetKey Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **net_idx**

Index of the NetKey

struct **esp_ble_mesh_cfg_app_key_status_cb_t**

Parameters of Config AppKey Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **net_idx**

Index of the NetKey

uint16_t **app_idx**

Index of the application key

struct **esp_ble_mesh_cfg_mod_app_status_cb_t**

Parameters of Config Model App Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **element_addr**

Address of the element

uint16_t **app_idx**

Index of the application key

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_cfg_friend_status_cb_t**

Parameter of Config Friend Status

Public Members

uint8_t **friend_state**

Friend state value

struct **esp_ble_mesh_cfg_hb_pub_status_cb_t**

Parameters of Config Heartbeat Publication Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **dst**

Destination address for Heartbeat messages

uint8_t **count**

Number of Heartbeat messages remaining to be sent

uint8_t **period**

Period for sending Heartbeat messages

uint8_t **ttl**

TTL to be used when sending Heartbeat messages

uint16_t **features**

Features that trigger Heartbeat messages when changed

uint16_t **net_idx**

Index of the NetKey

struct **esp_ble_mesh_cfg_hb_sub_status_cb_t**

Parameters of Config Heartbeat Subscription Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **src**

Source address for Heartbeat messages

uint16_t **dst**

Destination address for Heartbeat messages

uint8_t **period**

Remaining Period for processing Heartbeat messages

uint8_t **count**

Number of Heartbeat messages received

uint8_t **min_hops**

Minimum hops when receiving Heartbeat messages

uint8_t **max_hops**

Maximum hops when receiving Heartbeat messages

struct **esp_ble_mesh_cfg_net_trans_status_cb_t**

Parameters of Config Network Transmit Status

Public Members

uint8_t **net_trans_count**

Number of transmissions for each Network PDU originating from the node

uint8_t **net_trans_step**

Maximum hops when receiving Heartbeat messages

struct **esp_ble_mesh_cfg_model_sub_list_cb_t**

Parameters of Config SIG/Vendor Subscription List

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **element_addr**

Address of the element

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct net_buf_simple ***sub_addr**

A block of all addresses from the Subscription List

struct **esp_ble_mesh_cfg_net_key_list_cb_t**

Parameter of Config NetKey List

Public Members

struct net_buf_simple ***net_idx**

A list of NetKey Indexes known to the node

struct **esp_ble_mesh_cfg_app_key_list_cb_t**

Parameters of Config AppKey List

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **net_idx**

NetKey Index of the NetKey that the AppKeys are bound to

struct net_buf_simple ***app_idx**

A list of AppKey indexes that are bound to the NetKey identified by NetKeyIndex

struct **esp_ble_mesh_cfg_node_id_status_cb_t**

Parameters of Config Node Identity Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **net_idx**

Index of the NetKey

uint8_t **identity**

Node Identity state

struct **esp_ble_mesh_cfg_model_app_list_cb_t**

Parameters of Config SIG/Vendor Model App List

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **element_addr**

Address of the element

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct net_buf_simple ***app_idx**

All AppKey indexes bound to the Model

struct **esp_ble_mesh_cfg_kr_phase_status_cb_t**

Parameters of Config Key Refresh Phase Status

Public Members

uint8_t **status**

Status Code for the request message

uint16_t **net_idx**

Index of the NetKey

uint8_t **phase**

Key Refresh Phase state

struct **esp_ble_mesh_cfg_lpn_pollto_status_cb_t**

Parameters of Config Low Power Node PollTimeout Status

Public Members

uint16_t **lpn_addr**

The unicast address of the Low Power node

int32_t **poll_timeout**

The current value of the PollTimeout timer of the Low Power node

struct **esp_ble_mesh_cfg_client_cb_param_t**

Configuration Client Model callback parameters

Public Members

int **error_code**

Appropriate error code

esp_ble_mesh_client_common_param_t ***params**

The client common parameters

esp_ble_mesh_cfg_client_common_cb_param_t **status_cb**

The config status message callback values

struct **esp_ble_mesh_state_change_cfg_mod_pub_set_t**

Configuration Server model related context.

Public Members

uint16_t **element_addr**

Element Address

uint16_t **pub_addr**

Publish Address

uint16_t **app_idx**

AppKey Index

bool **cred_flag**

Friendship Credential Flag

uint8_t **pub_ttl**

Publish TTL

uint8_t **pub_period**

Publish Period

uint8_t **pub_retransmit**

Publish Retransmit

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_state_change_cfg_model_sub_add_t**

Parameters of Config Model Subscription Add

Public Members

uint16_t **element_addr**

Element Address

uint16_t **sub_addr**

Subscription Address

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_state_change_cfg_model_sub_delete_t**

Parameters of Config Model Subscription Delete

Public Members

uint16_t **element_addr**

Element Address

uint16_t **sub_addr**

Subscription Address

uint16_t **company_id**
Company ID

uint16_t **model_id**
Model ID

struct **esp_ble_mesh_state_change_cfg_netkey_add_t**
Parameters of Config NetKey Add

Public Members

uint16_t **net_idx**
NetKey Index

uint8_t **net_key**[16]
NetKey

struct **esp_ble_mesh_state_change_cfg_netkey_update_t**
Parameters of Config NetKey Update

Public Members

uint16_t **net_idx**
NetKey Index

uint8_t **net_key**[16]
NetKey

struct **esp_ble_mesh_state_change_cfg_netkey_delete_t**
Parameter of Config NetKey Delete

Public Members

uint16_t **net_idx**
NetKey Index

struct **esp_ble_mesh_state_change_cfg_appkey_add_t**
Parameters of Config AppKey Add

Public Members

uint16_t **net_idx**
NetKey Index

uint16_t **app_idx**
AppKey Index

uint8_t **app_key**[16]
AppKey

struct **esp_ble_mesh_state_change_cfg_appkey_update_t**
Parameters of Config AppKey Update

Public Members

uint16_t **net_idx**
NetKey Index

uint16_t **app_idx**
AppKey Index

uint8_t **app_key**[16]
AppKey

struct **esp_ble_mesh_state_change_cfg_appkey_delete_t**
Parameters of Config AppKey Delete

Public Members

uint16_t **net_idx**
NetKey Index

uint16_t **app_idx**
AppKey Index

struct **esp_ble_mesh_state_change_cfg_model_app_bind_t**
Parameters of Config Model App Bind

Public Members

uint16_t **element_addr**
Element Address

uint16_t **app_idx**
AppKey Index

uint16_t **company_id**
Company ID

uint16_t **model_id**
Model ID

struct **esp_ble_mesh_state_change_cfg_model_app_unbind_t**
Parameters of Config Model App Unbind

Public Members

uint16_t **element_addr**

Element Address

uint16_t **app_idx**

AppKey Index

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct **esp_ble_mesh_state_change_cfg_kr_phase_set_t**

Parameters of Config Key Refresh Phase Set

Public Members

uint16_t **net_idx**

NetKey Index

uint8_t **kr_phase**

New Key Refresh Phase Transition

struct **esp_ble_mesh_cfg_server_cb_param_t**

Configuration Server model callback parameters

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the server model structure

esp_ble_mesh_msg_ctx_t **ctx**

Context of the received message

esp_ble_mesh_cfg_server_cb_value_t **value**

Value of the received configuration messages

Macros

ESP_BLE_MESH_MODEL_CFG_SRV(srv_data)

Define a new Config Server Model.

Note: The Config Server Model can only be included by a Primary Element.

Parameters

- **srv_data** –Pointer to a unique Config Server Model user_data.

Returns New Config Server Model instance.

ESP_BLE_MESH_MODEL_CFG_CLI(cli_data)

Define a new Config Client Model.

Note: The Config Client Model can only be included by a Primary Element.

Parameters

- **cli_data** –Pointer to a unique struct *esp_ble_mesh_client_t*.

Returns New Config Client Model instance.

Type Definitions

typedef struct *esp_ble_mesh_cfg_srv* **esp_ble_mesh_cfg_srv_t**

Configuration Server Model context

typedef void (***esp_ble_mesh_cfg_client_cb_t**)(*esp_ble_mesh_cfg_client_cb_event_t* event, *esp_ble_mesh_cfg_client_cb_param_t* *param)

Bluetooth Mesh Config Client and Server Model functions.

Configuration Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (***esp_ble_mesh_cfg_server_cb_t**)(*esp_ble_mesh_cfg_server_cb_event_t* event, *esp_ble_mesh_cfg_server_cb_param_t* *param)

Configuration Server Model callback function type.

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum **esp_ble_mesh_cfg_client_cb_event_t**

This enum value is the event of Configuration Client Model

Values:

enumerator **ESP_BLE_MESH_CFG_CLIENT_GET_STATE_EVT**

enumerator **ESP_BLE_MESH_CFG_CLIENT_SET_STATE_EVT**

enumerator **ESP_BLE_MESH_CFG_CLIENT_PUBLISH_EVT**

enumerator **ESP_BLE_MESH_CFG_CLIENT_TIMEOUT_EVT**

enumerator **ESP_BLE_MESH_CFG_CLIENT_EVT_MAX**

enum **esp_ble_mesh_cfg_server_cb_event_t**

This enum value is the event of Configuration Server model

Values:

enumerator **ESP_BLE_MESH_CFG_SERVER_STATE_CHANGE_EVT**

enumerator **ESP_BLE_MESH_CFG_SERVER_EVT_MAX**

Health Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_health_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_health_client_callback**(*esp_ble_mesh_health_client_cb_t* callback)

Register BLE Mesh Health Model callback, the callback will report Health Client & Server Model events.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_health_server_callback**(*esp_ble_mesh_health_server_cb_t* callback)

Register BLE Mesh Health Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_health_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_health_client_get_state_t* *get_state)

This function is called to get the Health Server states using the Health Client Model get messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_opcode_health_client_get_t in esp_ble_mesh_defs.h

---

Parameters

- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_health_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_health_client_set_state_t* *set_state)

This function is called to set the Health Server states using the Health Client Model set messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_opcode_health_client_set_t in esp_ble_mesh_defs.h

---

Parameters

- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_health_server_fault_update**(*esp_ble_mesh_elem_t* *element)

This function is called by the Health Server Model to update the context of its Health Current status.

Parameters **element** –**[in]** The element to which the Health Server Model belongs.
Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_health_client_get_state_t**

#include <esp_ble_mesh_health_model_api.h> For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET ESP_BLE_MESH_MODEL_OP_ATTENTION_GET ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET the get_state parameter in the esp_ble_mesh_health_client_get_state function should not be set to NULL.

Public Members

esp_ble_mesh_health_fault_get_t **fault_get**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET.

union **esp_ble_mesh_health_client_set_state_t**

#include <esp_ble_mesh_health_model_api.h> For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK ESP_BLE_MESH_MODEL_OP_ATTENTION_SET ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK the set_state parameter in the esp_ble_mesh_health_client_set_state function should not be set to NULL.

Public Members

esp_ble_mesh_health_attention_set_t **attention_set**

For ESP_BLE_MESH_MODEL_OP_ATTENTION_SET or ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNA

esp_ble_mesh_health_period_set_t **period_set**

For ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET or ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK.

esp_ble_mesh_health_fault_test_t **fault_test**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST or ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK.

esp_ble_mesh_health_fault_clear_t **fault_clear**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR or ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK.

union **esp_ble_mesh_health_client_common_cb_param_t**

#include <esp_ble_mesh_health_model_api.h> Health Client Model received message union.

Public Members

esp_ble_mesh_health_current_status_cb_t **current_status**

The health current status value

esp_ble_mesh_health_fault_status_cb_t **fault_status**

The health fault status value

esp_ble_mesh_health_period_status_cb_t **period_status**

The health period status value

esp_ble_mesh_health_attention_status_cb_t **attention_status**

The health attention status value

union **esp_ble_mesh_health_server_cb_param_t**

#include <esp_ble_mesh_health_model_api.h> Health Server Model callback parameters union.

Public Members

esp_ble_mesh_health_fault_update_comp_cb_t **fault_update_comp**

ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT

esp_ble_mesh_health_fault_clear_cb_t **fault_clear**

ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT

esp_ble_mesh_health_fault_test_cb_t **fault_test**

ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT

esp_ble_mesh_health_attention_on_cb_t **attention_on**

ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT

esp_ble_mesh_health_attention_off_cb_t **attention_off**

ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT

Structures

struct **esp_ble_mesh_health_srv_cb_t**

ESP BLE Mesh Health Server callback

Public Members

esp_ble_mesh_cb_t **fault_clear**

Clear health registered faults. Initialized by the stack.

esp_ble_mesh_cb_t **fault_test**

Run a specific health test. Initialized by the stack.

esp_ble_mesh_cb_t **attention_on**

Health attention on callback. Initialized by the stack.

esp_ble_mesh_cb_t **attention_off**

Health attention off callback. Initialized by the stack.

struct **esp_ble_mesh_health_test_t**

ESP BLE Mesh Health Server test Context

Public Members

uint8_t **id_count**

Number of Health self-test ID

const uint8_t ***test_ids**

Array of Health self-test IDs

uint16_t **company_id**

Company ID used to identify the Health Fault state

uint8_t **prev_test_id**

Current test ID of the health fault test

uint8_t **current_faults**[ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE]

Array of current faults

uint8_t **registered_faults**[ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE]

Array of registered faults

struct **esp_ble_mesh_health_srv_t**

ESP BLE Mesh Health Server Model Context

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Health Server Model

esp_ble_mesh_health_srv_cb_t **health_cb**

Health callback struct

struct k_delayed_work **attention_timer**

Attention Timer state

bool **attention_timer_start**

Attention Timer start flag

esp_ble_mesh_health_test_t **health_test**

Health Server fault test

struct **esp_ble_mesh_health_fault_get_t**

Parameter of Health Fault Get

Public Members

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct **esp_ble_mesh_health_attention_set_t**

Parameter of Health Attention Set

Public Members

uint8_t **attention**

Value of the Attention Timer state

struct **esp_ble_mesh_health_period_set_t**

Parameter of Health Period Set

Public Members

uint8_t **fast_period_divisor**

Divider for the Publish Period

struct **esp_ble_mesh_health_fault_test_t**

Parameter of Health Fault Test

Public Members

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

uint8_t **test_id**

ID of a specific test to be performed

struct **esp_ble_mesh_health_fault_clear_t**

Parameter of Health Fault Clear

Public Members

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct **esp_ble_mesh_health_current_status_cb_t**

Parameters of Health Current Status

Public Members

uint8_t **test_id**

ID of a most recently performed test

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct net_buf_simple ***fault_array**

FaultArray field contains a sequence of 1-octet fault values

struct **esp_ble_mesh_health_fault_status_cb_t**

Parameters of Health Fault Status

Public Members

uint8_t **test_id**

ID of a most recently performed test

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct net_buf_simple ***fault_array**

FaultArray field contains a sequence of 1-octet fault values

struct **esp_ble_mesh_health_period_status_cb_t**

Parameter of Health Period Status

Public Members

uint8_t **fast_period_divisor**

Divider for the Publish Period

struct **esp_ble_mesh_health_attention_status_cb_t**

Parameter of Health Attention Status

Public Members

uint8_t **attention**

Value of the Attention Timer state

struct **esp_ble_mesh_health_client_cb_param_t**

Health Client Model callback parameters

Public Members

int **error_code**

Appropriate error code

esp_ble_mesh_client_common_param_t ***params**

The client common parameters.

esp_ble_mesh_health_client_common_cb_param_t **status_cb**

The health message status callback values

struct **esp_ble_mesh_health_fault_update_comp_cb_t**

Parameter of publishing Health Current Status completion event

Public Members

int **error_code**

The result of publishing Health Current Status

esp_ble_mesh_elem_t ***element**

Pointer to the element which contains the Health Server Model

struct **esp_ble_mesh_health_fault_clear_cb_t**

Parameters of Health Fault Clear event

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Health Server Model

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct **esp_ble_mesh_health_fault_test_cb_t**

Parameters of Health Fault Test event

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Health Server Model

uint8_t **test_id**

ID of a specific test to be performed

uint16_t **company_id**

Bluetooth assigned 16-bit Company ID

struct **esp_ble_mesh_health_attention_on_cb_t**

Parameter of Health Attention On event

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Health Server Model

uint8_t **time**

Duration of attention timer on (in seconds)

struct **esp_ble_mesh_health_attention_off_cb_t**

Parameter of Health Attention Off event

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Health Server Model

Macros

ESP_BLE_MESH_MODEL_HEALTH_SRV(srv, pub)

Define a new Health Server Model.

Note: The Health Server Model can only be included by a Primary Element.

Parameters

- **srv** –Pointer to the unique struct *esp_ble_mesh_health_srv_t*.
- **pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.

Returns New Health Server Model instance.

ESP_BLE_MESH_MODEL_HEALTH_CLI(cli_data)

Define a new Health Client Model.

Note: This API needs to be called for each element on which the application needs to have a Health Client Model.

Parameters

- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Health Client Model instance.

ESP_BLE_MESH_HEALTH_PUB_DEFINE(_name, _max, _role)

A helper to define a health publication context

Parameters

- **_name** –Name given to the publication context variable.
- **_max** –Maximum number of faults the element can have.
- **_role** –Role of the device which contains the model.

ESP_BLE_MESH_HEALTH_STANDARD_TEST

SIG identifier of Health Fault Test. 0x01 ~ 0xFF: Vendor Specific Test.

ESP_BLE_MESH_NO_FAULT

Fault values of Health Fault Test. 0x33 ~ 0x7F: Reserved for Future Use. 0x80 ~ 0xFF: Vendor Specific Warning/Error.

ESP_BLE_MESH_BATTERY_LOW_WARNING

ESP_BLE_MESH_BATTERY_LOW_ERROR

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_WARNING

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_ERROR

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_WARNING

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_ERROR

ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_WARNING

ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_ERROR

ESP_BLE_MESH_NO_LOAD_WARNING

ESP_BLE_MESH_NO_LOAD_ERROR

ESP_BLE_MESH_OVERLOAD_WARNING

ESP_BLE_MESH_OVERLOAD_ERROR

ESP_BLE_MESH_OVERHEAT_WARNING

ESP_BLE_MESH_OVERHEAT_ERROR

ESP_BLE_MESH_CONDENSATION_WARNING

ESP_BLE_MESH_CONDENSATION_ERROR

ESP_BLE_MESH_VIBRATION_WARNING

ESP_BLE_MESH_VIBRATION_ERROR

ESP_BLE_MESH_CONFIGURATION_WARNING

ESP_BLE_MESH_CONFIGURATION_ERROR

ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_WARNING

ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_ERROR

ESP_BLE_MESH_MEMORY_WARNING

ESP_BLE_MESH_MEMORY_ERROR

ESP_BLE_MESH_SELF_TEST_WARNING

ESP_BLE_MESH_SELF_TEST_ERROR

ESP_BLE_MESH_INPUT_TOO_LOW_WARNING

ESP_BLE_MESH_INPUT_TOO_LOW_ERROR

ESP_BLE_MESH_INPUT_TOO_HIGH_WARNING

ESP_BLE_MESH_INPUT_TOO_HIGH_ERROR

ESP_BLE_MESH_INPUT_NO_CHANGE_WARNING

ESP_BLE_MESH_INPUT_NO_CHANGE_ERROR

ESP_BLE_MESH_ACTUATOR_BLOCKED_WARNING

ESP_BLE_MESH_ACTUATOR_BLOCKED_ERROR

ESP_BLE_MESH_HOUSING_OPENED_WARNING

ESP_BLE_MESH_HOUSING_OPENED_ERROR

ESP_BLE_MESH_TAMPER_WARNING

ESP_BLE_MESH_TAMPER_ERROR

ESP_BLE_MESH_DEVICE_MOVED_WARNING

ESP_BLE_MESH_DEVICE_MOVED_ERROR

ESP_BLE_MESH_DEVICE_DROPPED_WARNING

ESP_BLE_MESH_DEVICE_DROPPED_ERROR

ESP_BLE_MESH_OVERFLOW_WARNING

ESP_BLE_MESH_OVERFLOW_ERROR

ESP_BLE_MESH_EMPTY_WARNING

ESP_BLE_MESH_EMPTY_ERROR

ESP_BLE_MESH_INTERNAL_BUS_WARNING

ESP_BLE_MESH_INTERNAL_BUS_ERROR

ESP_BLE_MESH_MECHANISM_JAMMED_WARNING

ESP_BLE_MESH_MECHANISM_JAMMED_ERROR

ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE

Type Definitions

typedef void (***esp_ble_mesh_health_client_cb_t**)(*esp_ble_mesh_health_client_cb_event_t* event, *esp_ble_mesh_health_client_cb_param_t* *param)

Bluetooth Mesh Health Client and Server Model function.

Health Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (***esp_ble_mesh_health_server_cb_t**)(*esp_ble_mesh_health_server_cb_event_t* event, *esp_ble_mesh_health_server_cb_param_t* *param)

Health Server Model callback function type.

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum **esp_ble_mesh_health_client_cb_event_t**

This enum value is the event of Health Client Model

Values:

enumerator **ESP_BLE_MESH_HEALTH_CLIENT_GET_STATE_EVT**

enumerator **ESP_BLE_MESH_HEALTH_CLIENT_SET_STATE_EVT**

enumerator **ESP_BLE_MESH_HEALTH_CLIENT_PUBLISH_EVT**

enumerator **ESP_BLE_MESH_HEALTH_CLIENT_TIMEOUT_EVT**

enumerator **ESP_BLE_MESH_HEALTH_CLIENT_EVT_MAX**

enum **esp_ble_mesh_health_server_cb_event_t**

This enum value is the event of Health Server Model

Values:

enumerator **ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT**

enumerator **ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT**

enumerator **ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT**

enumerator **ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT**

enumerator **ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT**

enumerator **ESP_BLE_MESH_HEALTH_SERVER_EVT_MAX**

Generic Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_generic_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_generic_client_callback**(*esp_ble_mesh_generic_client_cb_t* callback)

Register BLE Mesh Generic Client Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_generic_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_generic_client_get_state_t* *get_state)

Get the value of Generic Server Model states using the Generic Client Model get messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_generic_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer to generic get message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_generic_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_generic_client_set_state_t* *set_state)

Set the value of Generic Server Model states using the Generic Client Model set messages.

Note: If you want to find the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_generic_message_opcode_t in esp_ble_mesh_defs.h

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer to generic set message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_generic_server_callback**(*esp_ble_mesh_generic_server_cb_t* callback)

Register BLE Mesh Generic Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_generic_client_get_state_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Client Model get message union.

Public Members

esp_ble_mesh_gen_user_property_get_t **user_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET

esp_ble_mesh_gen_admin_property_get_t **admin_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET

esp_ble_mesh_gen_manufacturer_property_get_t **manufacturer_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET

esp_ble_mesh_gen_client_properties_get_t **client_properties_get**

For ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET

union **esp_ble_mesh_generic_client_set_state_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Client Model set message union.

Public Members

esp_ble_mesh_gen_onoff_set_t **onoff_set**

For ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET & ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UN

esp_ble_mesh_gen_level_set_t **level_set**

For ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET & ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UN

esp_ble_mesh_gen_delta_set_t **delta_set**

For ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET & ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UN

esp_ble_mesh_gen_move_set_t **move_set**

For ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET & ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNA

esp_ble_mesh_gen_def_trans_time_set_t **def_trans_time_set**

For ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET & ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNACK

esp_ble_mesh_gen_onpowerup_set_t **power_set**

For ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET & ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK

esp_ble_mesh_gen_power_level_set_t **power_level_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET & ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK

esp_ble_mesh_gen_power_default_set_t **power_default_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET & ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK

esp_ble_mesh_gen_power_range_set_t **power_range_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET & ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK

esp_ble_mesh_gen_loc_global_set_t **loc_global_set**

For ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET & ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK

esp_ble_mesh_gen_loc_local_set_t **loc_local_set**

For ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET & ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK

esp_ble_mesh_gen_user_property_set_t **user_property_set**

For ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET & ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK

esp_ble_mesh_gen_admin_property_set_t **admin_property_set**

For ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET & ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNACK

esp_ble_mesh_gen_manufacturer_property_set_t **manufacturer_property_set**

For ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET & ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET_UNACK

union **esp_ble_mesh_gen_client_status_cb_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Client Model received message union.

Public Members

esp_ble_mesh_gen_onoff_status_cb_t **onoff_status**

For ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS

esp_ble_mesh_gen_level_status_cb_t **level_status**

For ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS

esp_ble_mesh_gen_def_trans_time_status_cb_t **def_trans_time_status**

For ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS

esp_ble_mesh_gen_onpowerup_status_cb_t **onpowerup_status**

For ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS

esp_ble_mesh_gen_power_level_status_cb_t **power_level_status**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS

esp_ble_mesh_gen_power_last_status_cb_t **power_last_status**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS

esp_ble_mesh_gen_power_default_status_cb_t **power_default_status**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS

esp_ble_mesh_gen_power_range_status_cb_t **power_range_status**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS

esp_ble_mesh_gen_battery_status_cb_t **battery_status**

For ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS

esp_ble_mesh_gen_loc_global_status_cb_t **location_global_status**

For ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS

esp_ble_mesh_gen_loc_local_status_cb_t **location_local_status**

ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS

esp_ble_mesh_gen_user_properties_status_cb_t **user_properties_status**

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS

esp_ble_mesh_gen_user_property_status_cb_t **user_property_status**

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS

esp_ble_mesh_gen_admin_properties_status_cb_t **admin_properties_status**

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS

esp_ble_mesh_gen_admin_property_status_cb_t **admin_property_status**

ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS

esp_ble_mesh_gen_manufacturer_properties_status_cb_t **manufacturer_properties_status**

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_STATUS

esp_ble_mesh_gen_manufacturer_property_status_cb_t **manufacturer_property_status**

ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_STATUS

esp_ble_mesh_gen_client_properties_status_cb_t **client_properties_status**

ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATUS

union **esp_ble_mesh_generic_server_state_change_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Server Model state change value union.

Public Members

esp_ble_mesh_state_change_gen_onoff_set_t **onoff_set**

The recv_op in ctx can be used to decide which state is changed. Generic OnOff Set

esp_ble_mesh_state_change_gen_level_set_t **level_set**

Generic Level Set

esp_ble_mesh_state_change_gen_delta_set_t **delta_set**

Generic Delta Set

esp_ble_mesh_state_change_gen_move_set_t **move_set**

Generic Move Set

esp_ble_mesh_state_change_gen_def_trans_time_set_t **def_trans_time_set**

Generic Default Transition Time Set

esp_ble_mesh_state_change_gen_onpowerup_set_t **onpowerup_set**

Generic OnPowerUp Set

esp_ble_mesh_state_change_gen_power_level_set_t **power_level_set**

Generic Power Level Set

esp_ble_mesh_state_change_gen_power_default_set_t **power_default_set**

Generic Power Default Set

esp_ble_mesh_state_change_gen_power_range_set_t **power_range_set**

Generic Power Range Set

esp_ble_mesh_state_change_gen_loc_global_set_t **loc_global_set**

Generic Location Global Set

esp_ble_mesh_state_change_gen_loc_local_set_t **loc_local_set**

Generic Location Local Set

esp_ble_mesh_state_change_gen_user_property_set_t **user_property_set**

Generic User Property Set

esp_ble_mesh_state_change_gen_admin_property_set_t **admin_property_set**

Generic Admin Property Set

esp_ble_mesh_state_change_gen_manu_property_set_t **manu_property_set**

Generic Manufacturer Property Set

union **esp_ble_mesh_generic_server_recv_get_msg_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Server Model received get message union.

Public Members

esp_ble_mesh_server_recv_gen_user_property_get_t **user_property**

Generic User Property Get

esp_ble_mesh_server_recv_gen_admin_property_get_t **admin_property**

Generic Admin Property Get

esp_ble_mesh_server_recv_gen_manufacturer_property_get_t **manu_property**

Generic Manufacturer Property Get

esp_ble_mesh_server_recv_gen_client_properties_get_t **client_properties**

Generic Client Properties Get

union **esp_ble_mesh_generic_server_recv_set_msg_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Server Model received set message union.

Public Members

esp_ble_mesh_server_recv_gen_onoff_set_t **onoff**

Generic OnOff Set/Generic OnOff Set Unack

esp_ble_mesh_server_recv_gen_level_set_t **level**

Generic Level Set/Generic Level Set Unack

esp_ble_mesh_server_recv_gen_delta_set_t **delta**

Generic Delta Set/Generic Delta Set Unack

esp_ble_mesh_server_recv_gen_move_set_t **move**

Generic Move Set/Generic Move Set Unack

esp_ble_mesh_server_recv_gen_def_trans_time_set_t **def_trans_time**

Generic Default Transition Time Set/Generic Default Transition Time Set Unack

esp_ble_mesh_server_recv_gen_onpowerup_set_t **onpowerup**

Generic OnPowerUp Set/Generic OnPowerUp Set Unack

esp_ble_mesh_server_recv_gen_power_level_set_t **power_level**

Generic Power Level Set/Generic Power Level Set Unack

esp_ble_mesh_server_recv_gen_power_default_set_t **power_default**

Generic Power Default Set/Generic Power Default Set Unack

esp_ble_mesh_server_recv_gen_power_range_set_t **power_range**

Generic Power Range Set/Generic Power Range Set Unack

esp_ble_mesh_server_recv_gen_loc_global_set_t **location_global**

Generic Location Global Set/Generic Location Global Set Unack

esp_ble_mesh_server_recv_gen_loc_local_set_t **location_local**

Generic Location Local Set/Generic Location Local Set Unack

esp_ble_mesh_server_recv_gen_user_property_set_t **user_property**

Generic User Property Set/Generic User Property Set Unack

esp_ble_mesh_server_recv_gen_admin_property_set_t **admin_property**

Generic Admin Property Set/Generic Admin Property Set Unack

esp_ble_mesh_server_recv_gen_manufacturer_property_set_t **manu_property**

Generic Manufacturer Property Set/Generic Manufacturer Property Set Unack

union **esp_ble_mesh_generic_server_cb_value_t**

#include <esp_ble_mesh_generic_model_api.h> Generic Server Model callback value union.

Public Members

esp_ble_mesh_generic_server_state_change_t **state_change**

ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_generic_server_recv_get_msg_t **get**

ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_generic_server_recv_set_msg_t **set**

ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT

Structures

struct **esp_ble_mesh_gen_onoff_set_t**

Bluetooth Mesh Generic Client Model Get and Set parameters structure.

Parameters of Generic OnOff Set.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint8_t **onoff**

Target value of Generic OnOff state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_gen_level_set_t**

Parameters of Generic Level Set.

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **level**

Target value of Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_gen_delta_set_t**

Parameters of Generic Delta Set.

Public Members

bool **op_en**

Indicate if optional parameters are included

int32_t **level**

Delta change of Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_gen_move_set_t**

Parameters of Generic Move Set.

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **delta_level**

Delta Level step to calculate Move speed for Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_gen_def_trans_time_set_t**

Parameter of Generic Default Transition Time Set.

Public Members

uint8_t **trans_time**

The value of the Generic Default Transition Time state

struct **esp_ble_mesh_gen_onpowerup_set_t**

Parameter of Generic OnPowerUp Set.

Public Members

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct **esp_ble_mesh_gen_power_level_set_t**

Parameters of Generic Power Level Set.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **power**

Target value of Generic Power Actual state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_gen_power_default_set_t**

Parameter of Generic Power Default Set.

Public Members

uint16_t **power**

The value of the Generic Power Default state

struct **esp_ble_mesh_gen_power_range_set_t**

Parameters of Generic Power Range Set.

Public Members

uint16_t **range_min**

Value of Range Min field of Generic Power Range state

uint16_t **range_max**

Value of Range Max field of Generic Power Range state

struct **esp_ble_mesh_gen_loc_global_set_t**

Parameters of Generic Location Global Set.

Public Members

int32_t **global_latitude**

Global Coordinates (Latitude)

int32_t **global_longitude**

Global Coordinates (Longitude)

int16_t **global_altitude**

Global Altitude

struct **esp_ble_mesh_gen_loc_local_set_t**

Parameters of Generic Location Local Set.

Public Members

int16_t **local_north**
Local Coordinates (North)

int16_t **local_east**
Local Coordinates (East)

int16_t **local_altitude**
Local Altitude

uint8_t **floor_number**
Floor Number

uint16_t **uncertainty**
Uncertainty

struct **esp_ble_mesh_gen_user_property_get_t**
Parameter of Generic User Property Get.

Public Members

uint16_t **property_id**
Property ID identifying a Generic User Property

struct **esp_ble_mesh_gen_user_property_set_t**
Parameters of Generic User Property Set.

Public Members

uint16_t **property_id**
Property ID identifying a Generic User Property

struct net_buf_simple ***property_value**
Raw value for the User Property

struct **esp_ble_mesh_gen_admin_property_get_t**
Parameter of Generic Admin Property Get.

Public Members

uint16_t **property_id**
Property ID identifying a Generic Admin Property

struct **esp_ble_mesh_gen_admin_property_set_t**
Parameters of Generic Admin Property Set.

Public Members

uint16_t **property_id**

Property ID identifying a Generic Admin Property

uint8_t **user_access**

Enumeration indicating user access

struct net_buf_simple ***property_value**

Raw value for the Admin Property

struct **esp_ble_mesh_gen_manufacturer_property_get_t**

Parameter of Generic Manufacturer Property Get.

Public Members

uint16_t **property_id**

Property ID identifying a Generic Manufacturer Property

struct **esp_ble_mesh_gen_manufacturer_property_set_t**

Parameters of Generic Manufacturer Property Set.

Public Members

uint16_t **property_id**

Property ID identifying a Generic Manufacturer Property

uint8_t **user_access**

Enumeration indicating user access

struct **esp_ble_mesh_gen_client_properties_get_t**

Parameter of Generic Client Properties Get.

Public Members

uint16_t **property_id**

A starting Client Property ID present within an element

struct **esp_ble_mesh_gen_onoff_status_cb_t**

Bluetooth Mesh Generic Client Model Get and Set callback parameters structure.

Parameters of Generic OnOff Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint8_t **present_onoff**

Current value of Generic OnOff state

uint8_t **target_onoff**

Target value of Generic OnOff state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_gen_level_status_cb_t**

Parameters of Generic Level Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **present_level**

Current value of Generic Level state

int16_t **target_level**

Target value of the Generic Level state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_gen_def_trans_time_status_cb_t**

Parameter of Generic Default Transition Time Status.

Public Members

uint8_t **trans_time**

The value of the Generic Default Transition Time state

struct **esp_ble_mesh_gen_onpowerup_status_cb_t**

Parameter of Generic OnPowerUp Status.

Public Members

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct **esp_ble_mesh_gen_power_level_status_cb_t**

Parameters of Generic Power Level Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_power**

Current value of Generic Power Actual state

uint16_t **target_power**

Target value of Generic Power Actual state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_gen_power_last_status_cb_t**

Parameter of Generic Power Last Status.

Public Members

uint16_t **power**

The value of the Generic Power Last state

struct **esp_ble_mesh_gen_power_default_status_cb_t**

Parameter of Generic Power Default Status.

Public Members

uint16_t **power**

The value of the Generic Default Last state

struct **esp_ble_mesh_gen_power_range_status_cb_t**

Parameters of Generic Power Range Status.

Public Members

uint8_t **status_code**

Status Code for the request message

uint16_t **range_min**

Value of Range Min field of Generic Power Range state

uint16_t **range_max**

Value of Range Max field of Generic Power Range state

struct **esp_ble_mesh_gen_battery_status_cb_t**

Parameters of Generic Battery Status.

Public Members

uint32_t **battery_level**

Value of Generic Battery Level state

uint32_t **time_to_discharge**

Value of Generic Battery Time to Discharge state

uint32_t **time_to_charge**

Value of Generic Battery Time to Charge state

uint32_t **flags**

Value of Generic Battery Flags state

struct **esp_ble_mesh_gen_loc_global_status_cb_t**

Parameters of Generic Location Global Status.

Public Members

int32_t **global_latitude**

Global Coordinates (Latitude)

int32_t **global_longitude**

Global Coordinates (Longitude)

int16_t **global_altitude**

Global Altitude

struct **esp_ble_mesh_gen_loc_local_status_cb_t**

Parameters of Generic Location Local Status.

Public Members

int16_t **local_north**

Local Coordinates (North)

int16_t **local_east**

Local Coordinates (East)

int16_t **local_altitude**

Local Altitude

uint8_t **floor_number**

Floor Number

uint16_t **uncertainty**

Uncertainty

struct **esp_ble_mesh_gen_user_properties_status_cb_t**

Parameter of Generic User Properties Status.

Public Members

struct net_buf_simple ***property_ids**

Buffer contains a sequence of N User Property IDs

struct **esp_ble_mesh_gen_user_property_status_cb_t**

Parameters of Generic User Property Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID identifying a Generic User Property

uint8_t **user_access**

Enumeration indicating user access (optional)

struct net_buf_simple ***property_value**

Raw value for the User Property (C.1)

struct **esp_ble_mesh_gen_admin_properties_status_cb_t**

Parameter of Generic Admin Properties Status.

Public Members

struct net_buf_simple ***property_ids**

Buffer contains a sequence of N Admin Property IDs

struct **esp_ble_mesh_gen_admin_property_status_cb_t**

Parameters of Generic Admin Property Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID identifying a Generic Admin Property

uint8_t **user_access**

Enumeration indicating user access (optional)

struct net_buf_simple ***property_value**

Raw value for the Admin Property (C.1)

struct **esp_ble_mesh_gen_manufacturer_properties_status_cb_t**

Parameter of Generic Manufacturer Properties Status.

Public Members

struct net_buf_simple ***property_ids**

Buffer contains a sequence of N Manufacturer Property IDs

struct **esp_ble_mesh_gen_manufacturer_property_status_cb_t**

Parameters of Generic Manufacturer Property Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID identifying a Generic Manufacturer Property

uint8_t **user_access**

Enumeration indicating user access (optional)

struct net_buf_simple ***property_value**

Raw value for the Manufacturer Property (C.1)

struct **esp_ble_mesh_gen_client_properties_status_cb_t**

Parameter of Generic Client Properties Status.

Public Members

struct net_buf_simple ***property_ids**

Buffer contains a sequence of N Client Property IDs

struct **esp_ble_mesh_generic_client_cb_param_t**

Generic Client Model callback parameters

Public Members

int **error_code**

Appropriate error code

esp_ble_mesh_client_common_param_t ***params**

The client common parameters.

esp_ble_mesh_gen_client_status_cb_t **status_cb**

The generic status message callback values

struct **esp_ble_mesh_gen_onoff_state_t**

Parameters of Generic OnOff state

Public Members

uint8_t **onoff**

The present value of the Generic OnOff state

uint8_t **target_onoff**

The target value of the Generic OnOff state

struct **esp_ble_mesh_gen_onoff_srv_t**

User data of Generic OnOff Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic OnOff Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_onoff_state_t **state**

Parameters of the Generic OnOff state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

struct **esp_ble_mesh_gen_level_state_t**

Parameters of Generic Level state

Public Members

int16_t **level**

The present value of the Generic Level state

int16_t **target_level**

The target value of the Generic Level state

int16_t **last_level**

When a new transaction starts, level should be set to last_last, and use "level + incoming delta" to calculate the target level. In another word, "last_level" is used to record "level" of the last transaction, and "last_delta" is used to record the previously received delta_level value. The last value of the Generic Level state

int32_t **last_delta**

The last delta change of the Generic Level state

bool **move_start**

Indicate if the transition of the Generic Level state has been started

bool **positive**

Indicate if the transition is positive or negative

struct **esp_ble_mesh_gen_level_srv_t**

User data of Generic Level Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Level Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_level_state_t **state**

Parameters of the Generic Level state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_level**

Delta change value of level state transition

struct **esp_ble_mesh_gen_def_trans_time_state_t**

Parameter of Generic Default Transition Time state

Public Members

uint8_t **trans_time**

The value of the Generic Default Transition Time state

struct **esp_ble_mesh_gen_def_trans_time_srv_t**

User data of Generic Default Transition Time Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Default Transition Time Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_def_trans_time_state_t **state**

Parameters of the Generic Default Transition Time state

struct **esp_ble_mesh_gen_onpowerup_state_t**

Parameter of Generic OnPowerUp state

Public Members

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct **esp_ble_mesh_gen_power_onoff_srv_t**

User data of Generic Power OnOff Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power OnOff Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_onpowerup_state_t ***state**

Parameters of the Generic OnPowerUp state

struct **esp_ble_mesh_gen_power_onoff_setup_srv_t**

User data of Generic Power OnOff Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power OnOff Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_onpowerup_state_t ***state**

Parameters of the Generic OnPowerUp state

struct **esp_ble_mesh_gen_power_level_state_t**

Parameters of Generic Power Level state

Public Members

uint16_t **power_actual**

The present value of the Generic Power Actual state

uint16_t **target_power_actual**

The target value of the Generic Power Actual state

uint16_t **power_last**

The value of the Generic Power Last state

uint16_t **power_default**

The value of the Generic Power Default state

uint8_t **status_code**

The status code of setting Generic Power Range state

uint16_t **power_range_min**

The minimum value of the Generic Power Range state

uint16_t **power_range_max**

The maximum value of the Generic Power Range state

struct **esp_ble_mesh_gen_power_level_srv_t**

User data of Generic Power Level Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power Level Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_power_level_state_t ***state**

Parameters of the Generic Power Level state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_level**

Delta change value of level state transition

struct **esp_ble_mesh_gen_power_level_setup_srv_t**

User data of Generic Power Level Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power Level Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_power_level_state_t ***state**

Parameters of the Generic Power Level state

struct **esp_ble_mesh_gen_battery_state_t**

Parameters of Generic Battery state

Public Members

uint32_t **battery_level**

The value of the Generic Battery Level state

uint32_t **time_to_discharge**

The value of the Generic Battery Time to Discharge state

uint32_t **time_to_charge**

The value of the Generic Battery Time to Charge state

uint32_t **battery_flags**

The value of the Generic Battery Flags state

struct **esp_ble_mesh_gen_battery_srv_t**

User data of Generic Battery Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Battery Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_battery_state_t **state**

Parameters of the Generic Battery state

struct **esp_ble_mesh_gen_location_state_t**

Parameters of Generic Location state

Public Members

int32_t **global_latitude**

The value of the Global Latitude field

int32_t **global_longitude**

The value of the Global Longitude field

int16_t **global_altitude**

The value of the Global Altitude field

int16_t **local_north**

The value of the Local North field

int16_t **local_east**

The value of the Local East field

int16_t **local_altitude**

The value of the Local Altitude field

uint8_t **floor_number**

The value of the Floor Number field

uint16_t **uncertainty**

The value of the Uncertainty field

struct **esp_ble_mesh_gen_location_srv_t**

User data of Generic Location Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Location Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_location_state_t ***state**

Parameters of the Generic Location state

struct **esp_ble_mesh_gen_location_setup_srv_t**

User data of Generic Location Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Location Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_location_state_t ***state**

Parameters of the Generic Location state

struct **esp_ble_mesh_generic_property_t**

Parameters of Generic Property states

Public Members

uint16_t **id**

The value of User/Admin/Manufacturer Property ID

uint8_t **user_access**

The value of User Access field

uint8_t **admin_access**

The value of Admin Access field

uint8_t **manu_access**

The value of Manufacturer Access field

struct net_buf_simple ***val**

The value of User/Admin/Manufacturer Property

struct **esp_ble_mesh_gen_user_prop_srv_t**

User data of Generic User Property Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic User Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

uint8_t **property_count**

Generic User Property count

esp_ble_mesh_generic_property_t ***properties**

Parameters of the Generic User Property state

struct **esp_ble_mesh_gen_admin_prop_srv_t**

User data of Generic Admin Property Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Admin Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

uint8_t **property_count**

Generic Admin Property count

esp_ble_mesh_generic_property_t ***properties**

Parameters of the Generic Admin Property state

struct **esp_ble_mesh_gen_manu_prop_srv_t**

User data of Generic Manufacturer Property Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Manufacturer Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

uint8_t **property_count**

Generic Manufacturer Property count

esp_ble_mesh_generic_property_t ***properties**

Parameters of the Generic Manufacturer Property state

struct **esp_ble_mesh_gen_client_prop_srv_t**

User data of Generic Client Property Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Client Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

uint8_t **id_count**

Generic Client Property ID count

uint16_t ***property_ids**

Parameters of the Generic Client Property state

struct **esp_ble_mesh_state_change_gen_onoff_set_t**

Parameter of Generic OnOff Set state change event

Public Members

uint8_t **onoff**

The value of Generic OnOff state

struct **esp_ble_mesh_state_change_gen_level_set_t**

Parameter of Generic Level Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct **esp_ble_mesh_state_change_gen_delta_set_t**

Parameter of Generic Delta Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct **esp_ble_mesh_state_change_gen_move_set_t**

Parameter of Generic Move Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct **esp_ble_mesh_state_change_gen_def_trans_time_set_t**

Parameter of Generic Default Transition Time Set state change event

Public Members

uint8_t **trans_time**

The value of Generic Default Transition Time state

struct **esp_ble_mesh_state_change_gen_onpowerup_set_t**

Parameter of Generic OnPowerUp Set state change event

Public Members

uint8_t **onpowerup**

The value of Generic OnPowerUp state

struct **esp_ble_mesh_state_change_gen_power_level_set_t**

Parameter of Generic Power Level Set state change event

Public Members

uint16_t **power**

The value of Generic Power Actual state

struct **esp_ble_mesh_state_change_gen_power_default_set_t**

Parameter of Generic Power Default Set state change event

Public Members

uint16_t **power**

The value of Generic Power Default state

struct **esp_ble_mesh_state_change_gen_power_range_set_t**

Parameters of Generic Power Range Set state change event

Public Members

uint16_t **range_min**

The minimum value of Generic Power Range state

uint16_t **range_max**

The maximum value of Generic Power Range state

struct **esp_ble_mesh_state_change_gen_loc_global_set_t**

Parameters of Generic Location Global Set state change event

Public Members

int32_t **latitude**

The Global Latitude value of Generic Location state

int32_t **longitude**

The Global Longitude value of Generic Location state

int16_t **altitude**

The Global Altitude value of Generic Location state

struct **esp_ble_mesh_state_change_gen_loc_local_set_t**

Parameters of Generic Location Local Set state change event

Public Members

int16_t **north**

The Local North value of Generic Location state

int16_t **east**

The Local East value of Generic Location state

int16_t **altitude**

The Local Altitude value of Generic Location state

uint8_t **floor_number**

The Floor Number value of Generic Location state

uint16_t **uncertainty**

The Uncertainty value of Generic Location state

struct **esp_ble_mesh_state_change_gen_user_property_set_t**

Parameters of Generic User Property Set state change event

Public Members

uint16_t **id**

The property id of Generic User Property state

struct net_buf_simple ***value**

The property value of Generic User Property state

struct **esp_ble_mesh_state_change_gen_admin_property_set_t**

Parameters of Generic Admin Property Set state change event

Public Members

uint16_t **id**

The property id of Generic Admin Property state

uint8_t **access**

The property access of Generic Admin Property state

struct net_buf_simple ***value**

The property value of Generic Admin Property state

struct **esp_ble_mesh_state_change_gen_manu_property_set_t**

Parameters of Generic Manufacturer Property Set state change event

Public Members

uint16_t **id**

The property id of Generic Manufacturer Property state

uint8_t **access**

The property value of Generic Manufacturer Property state

struct **esp_ble_mesh_server_recv_gen_user_property_get_t**

Context of the received Generic User Property Get message

Public Members

uint16_t **property_id**

Property ID identifying a Generic User Property

struct **esp_ble_mesh_server_recv_gen_admin_property_get_t**

Context of the received Generic Admin Property Get message

Public Members

uint16_t **property_id**

Property ID identifying a Generic Admin Property

struct **esp_ble_mesh_server_recv_gen_manufacturer_property_get_t**

Context of the received Generic Manufacturer Property message

Public Members

uint16_t **property_id**

Property ID identifying a Generic Manufacturer Property

struct **esp_ble_mesh_server_recv_gen_client_properties_get_t**

Context of the received Generic Client Properties Get message

Public Members

uint16_t **property_id**

A starting Client Property ID present within an element

struct **esp_ble_mesh_server_recv_gen_onoff_set_t**

Context of the received Generic OnOff Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint8_t **onoff**

Target value of Generic OnOff state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_gen_level_set_t**

Context of the received Generic Level Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **level**

Target value of Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_gen_delta_set_t**

Context of the received Generic Delta Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

int32_t **delta_level**

Delta change of Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_gen_move_set_t**

Context of the received Generic Move Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **delta_level**

Delta Level step to calculate Move speed for Generic Level state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_gen_def_trans_time_set_t**

Context of the received Generic Default Transition Time Set message

Public Members

uint8_t **trans_time**

The value of the Generic Default Transition Time state

struct **esp_ble_mesh_server_recv_gen_onpowerup_set_t**

Context of the received Generic OnPowerUp Set message

Public Members

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct **esp_ble_mesh_server_recv_gen_power_level_set_t**

Context of the received Generic Power Level Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **power**

Target value of Generic Power Actual state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_gen_power_default_set_t**

Context of the received Generic Power Default Set message

Public Members

uint16_t **power**

The value of the Generic Power Default state

struct **esp_ble_mesh_server_recv_gen_power_range_set_t**

Context of the received Generic Power Range Set message

Public Members

uint16_t **range_min**

Value of Range Min field of Generic Power Range state

uint16_t **range_max**

Value of Range Max field of Generic Power Range state

struct **esp_ble_mesh_server_recv_gen_loc_global_set_t**

Context of the received Generic Location Global Set message

Public Members

int32_t **global_latitude**
Global Coordinates (Latitude)

int32_t **global_longitude**
Global Coordinates (Longitude)

int16_t **global_altitude**
Global Altitude

struct **esp_ble_mesh_server_recv_gen_loc_local_set_t**
Context of the received Generic Location Local Set message

Public Members

int16_t **local_north**
Local Coordinates (North)

int16_t **local_east**
Local Coordinates (East)

int16_t **local_altitude**
Local Altitude

uint8_t **floor_number**
Floor Number

uint16_t **uncertainty**
Uncertainty

struct **esp_ble_mesh_server_recv_gen_user_property_set_t**
Context of the received Generic User Property Set message

Public Members

uint16_t **property_id**
Property ID identifying a Generic User Property

struct net_buf_simple ***property_value**
Raw value for the User Property

struct **esp_ble_mesh_server_recv_gen_admin_property_set_t**
Context of the received Generic Admin Property Set message

Public Members

uint16_t **property_id**

Property ID identifying a Generic Admin Property

uint8_t **user_access**

Enumeration indicating user access

struct net_buf_simple ***property_value**

Raw value for the Admin Property

struct **esp_ble_mesh_server_recv_gen_manufacturer_property_set_t**

Context of the received Generic Manufacturer Property Set message

Public Members

uint16_t **property_id**

Property ID identifying a Generic Manufacturer Property

uint8_t **user_access**

Enumeration indicating user access

struct **esp_ble_mesh_generic_server_cb_param_t**

Generic Server Model callback parameters

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Generic Server Models

esp_ble_mesh_msg_ctx_t **ctx**

Context of the received messages

esp_ble_mesh_generic_server_cb_value_t **value**

Value of the received Generic Messages

Macros

ESP_BLE_MESH_MODEL_GEN_ONOFF_CLI(cli_pub, cli_data)

Define a new Generic OnOff Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic OnOff Client Model.

Parameters

- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic OnOff Client Model instance.

ESP_BLE_MESH_MODEL_GEN_LEVEL_CLI(cli_pub, cli_data)

Define a new Generic Level Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Level Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Level Client Model instance.

ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_CLI(cli_pub, cli_data)

Define a new Generic Default Transition Time Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Default Transition Time Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Default Transition Time Client Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_CLI(cli_pub, cli_data)

Define a new Generic Power OnOff Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Power OnOff Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Power OnOff Client Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_CLI(cli_pub, cli_data)

Define a new Generic Power Level Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Power Level Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Power Level Client Model instance.

ESP_BLE_MESH_MODEL_GEN_BATTERY_CLI(cli_pub, cli_data)

Define a new Generic Battery Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Battery Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Battery Client Model instance.

ESP_BLE_MESH_MODEL_GEN_LOCATION_CLI(cli_pub, cli_data)

Define a new Generic Location Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Location Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Location Client Model instance.

ESP_BLE_MESH_MODEL_GEN_PROPERTY_CLI(cli_pub, cli_data)

Define a new Generic Property Client Model.

Note: This API needs to be called for each element on which the application needs to have a Generic Property Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Generic Location Client Model instance.

ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV(srv_pub, srv_data)

Generic Server Models related context.

Define a new Generic OnOff Server Model.

Note: 1. The Generic OnOff Server Model is a root model.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_onoff_srv_t*.

Returns New Generic OnOff Server Model instance.

ESP_BLE_MESH_MODEL_GEN_LEVEL_SRV(srv_pub, srv_data)

Define a new Generic Level Server Model.

Note: 1. The Generic Level Server Model is a root model.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_level_srv_t*.

Returns New Generic Level Server Model instance.

ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_SRV(srv_pub, srv_data)

Define a new Generic Default Transition Time Server Model.

Note: 1. The Generic Default Transition Time Server Model is a root model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_def_trans_time_srv_t*.

Returns New Generic Default Transition Time Server Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SRV(srv_pub, srv_data)

Define a new Generic Power OnOff Server Model.

Note: 1. The Generic Power OnOff Server model extends the Generic OnOff Server model. When this model is present on an element, the corresponding Generic Power OnOff Setup Server model shall also be present.

a. This model may be used to represent a variety of devices that do not fit any of the model descriptions that have been defined but support the generic properties of On/Off.
b. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_power_onoff_srv_t*.

Returns New Generic Power OnOff Server Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SETUP_SRV(srv_pub, srv_data)

Define a new Generic Power OnOff Setup Server Model.

Note: 1. The Generic Power OnOff Setup Server model extends the Generic Power OnOff Server model and the Generic Default Transition Time Server model.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_power_onoff_setup_srv_t*.

Returns New Generic Power OnOff Setup Server Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SRV(srv_pub, srv_data)

Define a new Generic Power Level Server Model.

Note: 1. The Generic Power Level Server model extends the Generic Power OnOff Server model and the Generic Level Server model. When this model is present on an Element, the corresponding Generic Power Level Setup Server model shall also be present.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_power_level_srv_t*.

Returns New Generic Power Level Server Model instance.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SETUP_SRV (srv_pub, srv_data)

Define a new Generic Power Level Setup Server Model.

Note: 1. The Generic Power Level Setup Server model extends the Generic Power Level Server model and the Generic Power OnOff Setup Server model.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_power_level_setup_srv_t*.

Returns New Generic Power Level Setup Server Model instance.

ESP_BLE_MESH_MODEL_GEN_BATTERY_SRV (srv_pub, srv_data)

Define a new Generic Battery Server Model.

Note: 1. The Generic Battery Server Model is a root model.

a. This model shall support model publication and model subscription.
b. The model may be used to represent an element that is powered by a battery.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_battery_srv_t*.

Returns New Generic Battery Server Model instance.

ESP_BLE_MESH_MODEL_GEN_LOCATION_SRV (srv_pub, srv_data)

Define a new Generic Location Server Model.

Note: 1. The Generic Location Server model is a root model. When this model is present on an Element, the corresponding Generic Location Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. The model may be used to represent an element that knows its location (global or local).

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_location_srv_t*.

Returns New Generic Location Server Model instance.

ESP_BLE_MESH_MODEL_GEN_LOCATION_SETUP_SRV (srv_pub, srv_data)

Define a new Generic Location Setup Server Model.

Note: 1. The Generic Location Setup Server model extends the Generic Location Server model.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_location_setup_srv_t*.

Returns New Generic Location Setup Server Model instance.

ESP_BLE_MESH_MODEL_GEN_USER_PROP_SRV(srv_pub, srv_data)

Define a new Generic User Property Server Model.

Note: 1. The Generic User Property Server model is a root model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_user_prop_srv_t*.

Returns New Generic User Property Server Model instance.

ESP_BLE_MESH_MODEL_GEN_ADMIN_PROP_SRV(srv_pub, srv_data)

Define a new Generic Admin Property Server Model.

Note: 1. The Generic Admin Property Server model extends the Generic User Property Server model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_admin_prop_srv_t*.

Returns New Generic Admin Property Server Model instance.

ESP_BLE_MESH_MODEL_GEN_MANUFACTURER_PROP_SRV(srv_pub, srv_data)

Define a new Generic Manufacturer Property Server Model.

Note: 1. The Generic Manufacturer Property Server model extends the Generic User Property Server model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_manu_prop_srv_t*.

Returns New Generic Manufacturer Property Server Model instance.

ESP_BLE_MESH_MODEL_GEN_CLIENT_PROP_SRV(srv_pub, srv_data)

Define a new Generic User Property Server Model.

Note: 1. The Generic Client Property Server model is a root model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_gen_client_prop_srv_t*.

Returns New Generic Client Property Server Model instance.

Type Definitions

typedef void (*`esp_ble_mesh_generic_client_cb_t`)(*esp_ble_mesh_generic_client_cb_event_t* event, *esp_ble_mesh_generic_client_cb_param_t* *param)

Bluetooth Mesh Generic Client Model function.

Generic Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (*`esp_ble_mesh_generic_server_cb_t`)(*esp_ble_mesh_generic_server_cb_event_t* event, *esp_ble_mesh_generic_server_cb_param_t* *param)

Bluetooth Mesh Generic Server Model function.

Generic Server Model callback function type

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum `esp_ble_mesh_generic_client_cb_event_t`

This enum value is the event of Generic Client Model

Values:

enumerator `ESP_BLE_MESH_GENERIC_CLIENT_GET_STATE_EVT`

enumerator `ESP_BLE_MESH_GENERIC_CLIENT_SET_STATE_EVT`

enumerator `ESP_BLE_MESH_GENERIC_CLIENT_PUBLISH_EVT`

enumerator `ESP_BLE_MESH_GENERIC_CLIENT_TIMEOUT_EVT`

enumerator `ESP_BLE_MESH_GENERIC_CLIENT_EVT_MAX`

enum `esp_ble_mesh_gen_user_prop_access_t`

This enum value is the access value of Generic User Property

Values:

enumerator `ESP_BLE_MESH_GEN_USER_ACCESS_PROHIBIT`

enumerator `ESP_BLE_MESH_GEN_USER_ACCESS_READ`

enumerator `ESP_BLE_MESH_GEN_USER_ACCESS_WRITE`

enumerator `ESP_BLE_MESH_GEN_USER_ACCESS_READ_WRITE`

enum `esp_ble_mesh_gen_admin_prop_access_t`

This enum value is the access value of Generic Admin Property

Values:

enumerator `ESP_BLE_MESH_GEN_ADMIN_NOT_USER_PROP`

enumerator **ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ**

enumerator **ESP_BLE_MESH_GEN_ADMIN_ACCESS_WRITE**

enumerator **ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ_WRITE**

enum **esp_ble_mesh_gen_manu_prop_access_t**

This enum value is the access value of Generic Manufacturer Property

Values:

enumerator **ESP_BLE_MESH_GEN_MANU_NOT_USER_PROP**

enumerator **ESP_BLE_MESH_GEN_MANU_ACCESS_READ**

enum **esp_ble_mesh_generic_server_cb_event_t**

This enum value is the event of Generic Server Model

Values:

enumerator **ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT**

i. When get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, no event will be callback to the application layer when Generic Get messages are received.
ii. When set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, this event will be callback to the application layer when Generic Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT**

When get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Generic Get messages are received.

enumerator **ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT**

When set_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Generic Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_GENERIC_SERVER_EVT_MAX**

Sensor Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_sensor_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_sensor_client_callback**(*esp_ble_mesh_sensor_client_cb_t* callback)

Register BLE Mesh Sensor Client Model callback.

Parameters **callback** – **[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_sensor_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_sensor_client_get_state_t* *get_state)

Get the value of Sensor Server Model states using the Sensor Client Model get messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_sensor_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer to sensor get message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_sensor_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_sensor_client_set_state_t* *set_state)

Set the value of Sensor Server Model states using the Sensor Client Model set messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_sensor_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer to sensor set message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_sensor_server_callback**(*esp_ble_mesh_sensor_server_cb_t* callback)

Register BLE Mesh Sensor Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.

Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_sensor_client_get_state_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model get message union.

Public Members

esp_ble_mesh_sensor_descriptor_get_t **descriptor_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET

esp_ble_mesh_sensor_cadence_get_t **cadence_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET

esp_ble_mesh_sensor_settings_get_t **settings_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET

esp_ble_mesh_sensor_setting_get_t **setting_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET

esp_ble_mesh_sensor_get_t **sensor_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_GET

esp_ble_mesh_sensor_column_get_t **column_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET

esp_ble_mesh_sensor_series_get_t **series_get**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET

union **esp_ble_mesh_sensor_client_set_state_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model set message union.

Public Members

esp_ble_mesh_sensor_cadence_set_t **cadence_set**

For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET & ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK

esp_ble_mesh_sensor_setting_set_t **setting_set**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET & ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK

union **esp_ble_mesh_sensor_client_status_cb_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model received message union.

Public Members

esp_ble_mesh_sensor_descriptor_status_cb_t **descriptor_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS

esp_ble_mesh_sensor_cadence_status_cb_t **cadence_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS

esp_ble_mesh_sensor_settings_status_cb_t **settings_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS

esp_ble_mesh_sensor_setting_status_cb_t **setting_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS

esp_ble_mesh_sensor_status_cb_t **sensor_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS

esp_ble_mesh_sensor_column_status_cb_t **column_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS

esp_ble_mesh_sensor_series_status_cb_t **series_status**

For ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS

union **esp_ble_mesh_sensor_server_state_change_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model state change value union.

Public Members

esp_ble_mesh_state_change_sensor_cadence_set_t **sensor_cadence_set**

The recv_op in ctx can be used to decide which state is changed. Sensor Cadence Set

esp_ble_mesh_state_change_sensor_setting_set_t **sensor_setting_set**

Sensor Setting Set

union **esp_ble_mesh_sensor_server_recv_get_msg_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model received get message union.

Public Members

esp_ble_mesh_server_recv_sensor_descriptor_get_t **sensor_descriptor**

Sensor Descriptor Get

esp_ble_mesh_server_recv_sensor_cadence_get_t **sensor_cadence**

Sensor Cadence Get

esp_ble_mesh_server_recv_sensor_settings_get_t **sensor_settings**

Sensor Settings Get

esp_ble_mesh_server_recv_sensor_setting_get_t **sensor_setting**

Sensor Setting Get

esp_ble_mesh_server_recv_sensor_get_t **sensor_data**

Sensor Get

esp_ble_mesh_server_recv_sensor_column_get_t **sensor_column**

Sensor Column Get

esp_ble_mesh_server_recv_sensor_series_get_t **sensor_series**

Sensor Series Get

union **esp_ble_mesh_sensor_server_recv_set_msg_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model received set message union.

Public Members

esp_ble_mesh_server_recv_sensor_cadence_set_t **sensor_cadence**

Sensor Cadence Set

esp_ble_mesh_server_recv_sensor_setting_set_t **sensor_setting**

Sensor Setting Set

union **esp_ble_mesh_sensor_server_cb_value_t**

#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model callback value union.

Public Members

esp_ble_mesh_sensor_server_state_change_t **state_change**

ESP_BLE_MESH_SENSOR_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_sensor_server_recv_get_msg_t **get**

ESP_BLE_MESH_SENSOR_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_sensor_server_recv_set_msg_t **set**

ESP_BLE_MESH_SENSOR_SERVER_RECV_SET_MSG_EVT

Structures

struct **esp_ble_mesh_sensor_descriptor_get_t**

Bluetooth Mesh Sensor Client Model Get and Set parameters structure.

Parameters of Sensor Descriptor Get

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID of a sensor (optional)

struct **esp_ble_mesh_sensor_cadence_get_t**

Parameter of Sensor Cadence Get

Public Members

uint16_t **property_id**

Property ID of a sensor

struct **esp_ble_mesh_sensor_cadence_set_t**

Parameters of Sensor Cadence Set

Public Members

uint16_t **property_id**

Property ID for the sensor

uint8_t **fast_cadence_period_divisor**

Divisor for the publish period

uint8_t **status_trigger_type**

The unit and format of the Status Trigger Delta fields

struct net_buf_simple ***status_trigger_delta_down**

Delta down value that triggers a status message

struct net_buf_simple ***status_trigger_delta_up**

Delta up value that triggers a status message

uint8_t **status_min_interval**

Minimum interval between two consecutive Status messages

struct net_buf_simple ***fast_cadence_low**

Low value for the fast cadence range

struct net_buf_simple ***fast_cadence_high**

Fast value for the fast cadence range

struct **esp_ble_mesh_sensor_settings_get_t**

Parameter of Sensor Settings Get

Public Members

uint16_t **sensor_property_id**

Property ID of a sensor

struct **esp_ble_mesh_sensor_setting_get_t**

Parameters of Sensor Setting Get

Public Members

uint16_t **sensor_property_id**

Property ID of a sensor

uint16_t **sensor_setting_property_id**

Setting ID identifying a setting within a sensor

struct **esp_ble_mesh_sensor_setting_set_t**

Parameters of Sensor Setting Set

Public Members

uint16_t **sensor_property_id**

Property ID identifying a sensor

uint16_t **sensor_setting_property_id**

Setting ID identifying a setting within a sensor

struct net_buf_simple ***sensor_setting_raw**

Raw value for the setting

struct **esp_ble_mesh_sensor_get_t**

Parameters of Sensor Get

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID for the sensor (optional)

struct **esp_ble_mesh_sensor_column_get_t**

Parameters of Sensor Column Get

Public Members

uint16_t **property_id**

Property identifying a sensor

struct net_buf_simple ***raw_value_x**

Raw value identifying a column

struct **esp_ble_mesh_sensor_series_get_t**

Parameters of Sensor Series Get

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property identifying a sensor

struct net_buf_simple ***raw_value_x1**

Raw value identifying a starting column (optional)

struct net_buf_simple ***raw_value_x2**

Raw value identifying an ending column (C.1)

struct **esp_ble_mesh_sensor_descriptor_status_cb_t**

Bluetooth Mesh Sensor Client Model Get and Set callback parameters structure.

Parameter of Sensor Descriptor Status

Public Members

struct net_buf_simple ***descriptor**

Sequence of 8-octet sensor descriptors (optional)

struct **esp_ble_mesh_sensor_cadence_status_cb_t**

Parameters of Sensor Cadence Status

Public Members

uint16_t **property_id**

Property for the sensor

struct net_buf_simple ***sensor_cadence_value**

Value of sensor cadence state

struct **esp_ble_mesh_sensor_settings_status_cb_t**

Parameters of Sensor Settings Status

Public Members

uint16_t **sensor_property_id**

Property ID identifying a sensor

struct net_buf_simple ***sensor_setting_property_ids**

A sequence of N sensor setting property IDs (optional)

struct **esp_ble_mesh_sensor_setting_status_cb_t**

Parameters of Sensor Setting Status

Public Members

bool **op_en**

Indicate id optional parameters are included

uint16_t **sensor_property_id**

Property ID identifying a sensor

uint16_t **sensor_setting_property_id**

Setting ID identifying a setting within a sensor

uint8_t **sensor_setting_access**

Read/Write access rights for the setting (optional)

struct net_buf_simple ***sensor_setting_raw**

Raw value for the setting

struct **esp_ble_mesh_sensor_status_cb_t**

Parameter of Sensor Status

Public Members

struct net_buf_simple ***marshalled_sensor_data**

Value of sensor data state (optional)

struct **esp_ble_mesh_sensor_column_status_cb_t**

Parameters of Sensor Column Status

Public Members

uint16_t **property_id**

Property identifying a sensor and the Y axis

struct net_buf_simple ***sensor_column_value**

Left values of sensor column status

struct **esp_ble_mesh_sensor_series_status_cb_t**

Parameters of Sensor Series Status

Public Members

uint16_t **property_id**

Property identifying a sensor and the Y axis

struct net_buf_simple ***sensor_series_value**

Left values of sensor series status

struct **esp_ble_mesh_sensor_client_cb_param_t**

Sensor Client Model callback parameters

Public Members

int **error_code**

0: success, otherwise failure. For the error code values please refer to errno.h file. A negative sign is added to the standard error codes in errno.h.

esp_ble_mesh_client_common_param_t ***params**

The client common parameters.

esp_ble_mesh_sensor_client_status_cb_t **status_cb**

The sensor status message callback values

struct **esp_ble_mesh_sensor_descriptor_t**

Parameters of Sensor Descriptor state

Public Members

uint32_t **positive_tolerance**

The value of Sensor Positive Tolerance field

uint32_t **negative_tolerance**

The value of Sensor Negative Tolerance field

uint32_t **sampling_function**

The value of Sensor Sampling Function field

uint8_t **measure_period**

The value of Sensor Measurement Period field

uint8_t **update_interval**

The value of Sensor Update Interval field

struct **esp_ble_mesh_sensor_setting_t**

Parameters of Sensor Setting state

Public Members

uint16_t **property_id**

The value of Sensor Setting Property ID field

uint8_t **access**

The value of Sensor Setting Access field

struct net_buf_simple ***raw**

The value of Sensor Setting Raw field

struct **esp_ble_mesh_sensor_cadence_t**

Parameters of Sensor Cadence state

Public Members

uint8_t **period_divisor**

The value of Fast Cadence Period Divisor field

uint8_t **trigger_type**

The value of Status Trigger Type field

struct net_buf_simple ***trigger_delta_down**

Note: The parameter "size" in trigger_delta_down, trigger_delta_up, fast_cadence_low & fast_cadence_high indicates the exact length of these four parameters, and they are associated with the Sensor Property ID. Users need to initialize the "size" precisely. The value of Status Trigger Delta Down field

struct net_buf_simple ***trigger_delta_up**

The value of Status Trigger Delta Up field

uint8_t **min_interval**

The value of Status Min Interval field

struct net_buf_simple ***fast_cadence_low**

The value of Fast Cadence Low field

struct net_buf_simple ***fast_cadence_high**

The value of Fast Cadence High field

struct **esp_ble_mesh_sensor_data_t**

Parameters of Sensor Data state

Public Members

uint8_t **format**

Format A: The Length field is a 1-based uint4 value (valid range 0x0–0xF, representing range of 1 –16). Format B: The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1 – 127). The value 0x7F represents a length of zero. The value of the Sensor Data format

uint8_t **length**

The value of the Sensor Data length

struct net_buf_simple ***raw_value**

The value of Sensor Data raw value

struct **esp_ble_mesh_sensor_series_column_t**

Parameters of Sensor Series Column state

Public Members

struct net_buf_simple ***raw_value_x**

The value of Sensor Raw Value X field

struct net_buf_simple ***column_width**

The value of Sensor Column Width field

struct net_buf_simple ***raw_value_y**

The value of Sensor Raw Value Y field

struct **esp_ble_mesh_sensor_state_t**

Parameters of Sensor states

Public Members

uint16_t **sensor_property_id**

The value of Sensor Property ID field

esp_ble_mesh_sensor_descriptor_t **descriptor**

Parameters of the Sensor Descriptor state

const uint8_t **setting_count**

Multiple Sensor Setting states may be present for each sensor. The Sensor Setting Property ID values shall be unique for each Sensor Property ID that identifies a sensor within an element.

esp_ble_mesh_sensor_setting_t ***settings**

Parameters of the Sensor Setting state

esp_ble_mesh_sensor_cadence_t ***cadence**

The Sensor Cadence state may be not supported by sensors based on device properties referencing "non-scalar characteristics" such as "histograms" or "composite characteristics" . Parameters of the Sensor Cadence state

esp_ble_mesh_sensor_data_t **sensor_data**

Parameters of the Sensor Data state

esp_ble_mesh_sensor_series_column_t **series_column**

Parameters of the Sensor Series Column state

struct **esp_ble_mesh_sensor_srv_t**

User data of Sensor Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Sensor Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

const uint8_t **state_count**

Sensor state count

esp_ble_mesh_sensor_state_t ***states**

Parameters of the Sensor states

struct **esp_ble_mesh_sensor_setup_srv_t**

User data of Sensor Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Sensor Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

const uint8_t **state_count**

Sensor state count

esp_ble_mesh_sensor_state_t ***states**

Parameters of the Sensor states

struct **esp_ble_mesh_state_change_sensor_cadence_set_t**

Parameters of Sensor Cadence Set state change event

Public Members

uint16_t **property_id**

The value of Sensor Property ID state

uint8_t **period_divisor**

The value of Fast Cadence Period Divisor state

uint8_t **trigger_type**

The value of Status Trigger Type state

struct net_buf_simple ***trigger_delta_down**

The value of Status Trigger Delta Down state

struct net_buf_simple ***trigger_delta_up**

The value of Status Trigger Delta Up state

uint8_t **min_interval**

The value of Status Min Interval state

struct net_buf_simple ***fast_cadence_low**

The value of Fast Cadence Low state

struct net_buf_simple ***fast_cadence_high**

The value of Fast Cadence High state

struct **esp_ble_mesh_state_change_sensor_setting_set_t**

Parameters of Sensor Setting Set state change event

Public Members

uint16_t **property_id**

The value of Sensor Property ID state

uint16_t **setting_property_id**

The value of Sensor Setting Property ID state

struct net_buf_simple ***setting_value**

The value of Sensor Property Value state

struct **esp_ble_mesh_server_recv_sensor_descriptor_get_t**

Context of the received Sensor Descriptor Get message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID of a sensor (optional)

struct **esp_ble_mesh_server_recv_sensor_cadence_get_t**

Context of the received Sensor Cadence Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

struct **esp_ble_mesh_server_recv_sensor_settings_get_t**

Context of the received Sensor Settings Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

struct **esp_ble_mesh_server_recv_sensor_setting_get_t**

Context of the received Sensor Setting Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

uint16_t **setting_property_id**

Setting ID identifying a setting within a sensor

struct **esp_ble_mesh_server_recv_sensor_get_t**

Context of the received Sensor Get message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID for the sensor (optional)

struct **esp_ble_mesh_server_recv_sensor_column_get_t**

Context of the received Sensor Column Get message

Public Members

uint16_t **property_id**

Property identifying a sensor

struct net_buf_simple ***raw_value_x**

Raw value identifying a column

struct **esp_ble_mesh_server_recv_sensor_series_get_t**

Context of the received Sensor Series Get message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property identifying a sensor

struct net_buf_simple ***raw_value**

Raw value containing X1 and X2 (optional)

struct **esp_ble_mesh_server_recv_sensor_cadence_set_t**

Context of the received Sensor Cadence Set message

Public Members

uint16_t **property_id**

Property ID for the sensor

struct net_buf_simple ***cadence**

Value of Sensor Cadence state

struct **esp_ble_mesh_server_recv_sensor_setting_set_t**

Context of the received Sensor Setting Set message

Public Members

uint16_t **property_id**

Property ID identifying a sensor

uint16_t **setting_property_id**

Setting ID identifying a setting within a sensor

struct net_buf_simple ***setting_raw**

Raw value for the setting

struct **esp_ble_mesh_sensor_server_cb_param_t**

Sensor Server Model callback parameters

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Sensor Server Models

esp_ble_mesh_msg_ctx_t **ctx**

Context of the received messages

esp_ble_mesh_sensor_server_cb_value_t **value**

Value of the received Sensor Messages

Macros

ESP_BLE_MESH_MODEL_SENSOR_CLI(cli_pub, cli_data)

Define a new Sensor Client Model.

Note: This API needs to be called for each element on which the application needs to have a Sensor Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Sensor Client Model instance.

ESP_BLE_MESH_MODEL_SENSOR_SRV(srv_pub, srv_data)

Sensor Server Models related context.

Define a new Sensor Server Model.

Note: 1. The Sensor Server model is a root model. When this model is present on an element, the corresponding Sensor Setup Server model shall also be present.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_sensor_srv_t*.

Returns New Sensor Server Model instance.

ESP_BLE_MESH_MODEL_SENSOR_SETUP_SRV(srv_pub, srv_data)

Define a new Sensor Setup Server Model.

Note: 1. The Sensor Setup Server model extends the Sensor Server model.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_sensor_setup_srv_t*.

Returns New Sensor Setup Server Model instance.

ESP_BLE_MESH_INVALID_SENSOR_PROPERTY_ID

Invalid Sensor Property ID

ESP_BLE_MESH_SENSOR_PROPERTY_ID_LEN

Length of Sensor Property ID

ESP_BLE_MESH_SENSOR_DESCRIPTOR_LEN

Length of Sensor Descriptor state

ESP_BLE_MESH_SENSOR_UNSPECIFIED_POS_TOLERANCE

Unspecified Sensor Positive Tolerance

ESP_BLE_MESH_SENSOR_UNSPECIFIED_NEG_TOLERANCE

Unspecified Sensor Negative Tolerance

ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD

Not applicable Sensor Measurement Period

ESP_BLE_MESH_SENSOR_NOT_APPL_UPDATE_INTERVAL

Not applicable Sensor Update Interval

ESP_BLE_MESH_INVALID_SENSOR_SETTING_PROPERTY_ID

Invalid Sensor Setting Property ID

ESP_BLE_MESH_SENSOR_SETTING_PROPERTY_ID_LEN

Length of Sensor Setting Property ID

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_LEN

Length of Sensor Setting Access

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ

Sensor Setting Access - Read

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ_WRITE

Sensor Setting Access - Read & Write

ESP_BLE_MESH_SENSOR_DIVISOR_TRIGGER_TYPE_LEN

Length of Sensor Divisor Trigger Type

ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_LEN

Length of Sensor Status Min Interval

ESP_BLE_MESH_SENSOR_PERIOD_DIVISOR_MAX_VALUE

Maximum value of Sensor Period Divisor

ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_MAX

Maximum value of Sensor Status Min Interval

ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_CHAR

Sensor Status Trigger Type - Format Type of the characteristic that the Sensor Property ID state references

ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_UINT16

Sensor Status Trigger Type - Format Type "uint16"

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A

Sensor Data Format A

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B

Sensor Data Format B

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID_LEN

MPID length of Sensor Data Format A

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN

MPID length of Sensor Data Format B

ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN

Zero length of Sensor Data.

Note: The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1–127). The value 0x7F represents a length of zero.

ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT(_data)

Get format of the sensor data.

Note: Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting the format of the corresponding sensor data.

Parameters

- **_data** –Pointer to the start of the sensor data.

Returns Format of the sensor data.

ESP_BLE_MESH_GET_SENSOR_DATA_LENGTH(_data, _fmt)

Get length of the sensor data.

Note: Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting the length of the corresponding sensor data.

Parameters

- **_data** –Pointer to the start of the sensor data.
- **_fmt** –Format of the sensor data.

Returns Length (zero-based) of the sensor data.

ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID(_data, _fmt)

Get Sensor Property ID of the sensor data.

Note: Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting Sensor Property ID of the corresponding sensor data.

Parameters

- **_data** –Pointer to the start of the sensor data.
- **_fmt** –Format of the sensor data.

Returns Sensor Property ID of the sensor data.

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID(_len, _id)

Generate a MPID value for sensor data with Format A.

Note: 1. The Format field is 0b0 and indicates that Format A is used.

a. The Length field is a 1-based uint4 value (valid range 0x0–0xF, representing range of 1–16).
b. The Property ID is an 11-bit bit field representing 11 LSb of a Property ID.
c. This format may be used for Property Values that are not longer than 16 octets and for Property IDs less than 0x0800.

Parameters

- **_len** –Length of Sensor Raw value.
- **_id** –Sensor Property ID.

Returns 2-octet MPID value for sensor data with Format A.

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID(_len, _id)

Generate a MPID value for sensor data with Format B.

Note: 1. The Format field is 0b1 and indicates Format B is used.

a. The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1–127). The value 0x7F represents a length of zero.
b. The Property ID is a 16-bit bit field representing a Property ID.
c. This format may be used for Property Values not longer than 128 octets and for any Property IDs. Property values longer than 128 octets are not supported by the Sensor Status message.
d. Exclude the generated 1-octet value, the 2-octet Sensor Property ID

Parameters
- **`_len`** –Length of Sensor Raw value.
- **`_id`** –Sensor Property ID.

Returns 3-octet MPID value for sensor data with Format B.

Type Definitions

typedef void (***esp_ble_mesh_sensor_client_cb_t**)(*esp_ble_mesh_sensor_client_cb_event_t* event, *esp_ble_mesh_sensor_client_cb_param_t* *param)

Bluetooth Mesh Sensor Client Model function.

Sensor Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (***esp_ble_mesh_sensor_server_cb_t**)(*esp_ble_mesh_sensor_server_cb_event_t* event, *esp_ble_mesh_sensor_server_cb_param_t* *param)

Bluetooth Mesh Sensor Server Model function.

Sensor Server Model callback function type

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum **esp_ble_mesh_sensor_client_cb_event_t**

This enum value is the event of Sensor Client Model

Values:

enumerator **ESP_BLE_MESH_SENSOR_CLIENT_GET_STATE_EVT**

enumerator **ESP_BLE_MESH_SENSOR_CLIENT_SET_STATE_EVT**

enumerator **ESP_BLE_MESH_SENSOR_CLIENT_PUBLISH_EVT**

enumerator **ESP_BLE_MESH_SENSOR_CLIENT_TIMEOUT_EVT**

enumerator **ESP_BLE_MESH_SENSOR_CLIENT_EVT_MAX**

enum **esp_ble_mesh_sensor_sample_func**

This enum value is value of Sensor Sampling Function

Values:

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_UNSPECIFIED**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_INSTANTANEOUS**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_ARITHMETIC_MEAN**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_RMS**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_MAXIMUM**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_MINIMUM**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_ACCUMULATED**

enumerator **ESP_BLE_MESH_SAMPLE_FUNC_COUNT**

enum **esp_ble_mesh_sensor_server_cb_event_t**

This enum value is the event of Sensor Server Model

Values:

enumerator **ESP_BLE_MESH_SENSOR_SERVER_STATE_CHANGE_EVT**

i. When get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, no event will be callback to the application layer when Sensor Get messages are received.
ii. When set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, this event will be callback to the application layer when Sensor Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_SENSOR_SERVER_RECV_GET_MSG_EVT**

When get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Sensor Get messages are received.

enumerator **ESP_BLE_MESH_SENSOR_SERVER_RECV_SET_MSG_EVT**

When set_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Sensor Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_SENSOR_SERVER_EVT_MAX**

Time and Scenes Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_time_scene_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_time_scene_client_callback**(*esp_ble_mesh_time_scene_client_cb_t* callback)

Register BLE Mesh Time Scene Client Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_time_scene_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_time_scene_client_get_state_t* *get_state)

Get the value of Time Scene Server Model states using the Time Scene Client Model get messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_time_scene_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters

- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer to time scene get message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_time_scene_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_time_scene_client_set_state_t* *set_state)

Set the value of Time Scene Server Model states using the Time Scene Client Model set messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_time_scene_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters

- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer to time scene set message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_time_scene_server_callback**(*esp_ble_mesh_time_scene_server_cb_t* callback)

Register BLE Mesh Time and Scenes Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.

Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_time_scene_client_get_state_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model get message union.

Public Members

esp_ble_mesh_scheduler_act_get_t **scheduler_act_get**

For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET

union **esp_ble_mesh_time_scene_client_set_state_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model set message union.

Public Members

esp_ble_mesh_time_set_t **time_set**

For ESP_BLE_MESH_MODEL_OP_TIME_SET

esp_ble_mesh_time_zone_set_t **time_zone_set**

For ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET

esp_ble_mesh_tai_utc_delta_set_t **tai_utc_delta_set**

For ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_SET

esp_ble_mesh_time_role_set_t **time_role_set**

For ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET

esp_ble_mesh_scene_store_t **scene_store**

For ESP_BLE_MESH_MODEL_OP_SCENE_STORE & ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK

esp_ble_mesh_scene_recall_t **scene_recall**

For ESP_BLE_MESH_MODEL_OP_SCENE_RECALL & ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNA

esp_ble_mesh_scene_delete_t **scene_delete**

For ESP_BLE_MESH_MODEL_OP_SCENE_DELETE & ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNAC

esp_ble_mesh_scheduler_act_set_t **scheduler_act_set**

For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET & ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK

union **esp_ble_mesh_time_scene_client_status_cb_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model received message union.

Public Members

esp_ble_mesh_time_status_cb_t **time_status**

For ESP_BLE_MESH_MODEL_OP_TIME_STATUS

esp_ble_mesh_time_zone_status_cb_t **time_zone_status**

For ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS

esp_ble_mesh_tai_utc_delta_status_cb_t **tai_utc_delta_status**

For ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_STATUS

esp_ble_mesh_time_role_status_cb_t **time_role_status**

For ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS

esp_ble_mesh_scene_status_cb_t **scene_status**

For ESP_BLE_MESH_MODEL_OP_SCENE_STATUS

esp_ble_mesh_scene_register_status_cb_t **scene_register_status**

For ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS

esp_ble_mesh_scheduler_status_cb_t **scheduler_status**

For ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS

esp_ble_mesh_scheduler_act_status_cb_t **scheduler_act_status**

For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS

union **esp_ble_mesh_time_scene_server_state_change_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model state change value union.

Public Members

esp_ble_mesh_state_change_time_set_t **time_set**

The recv_op in ctx can be used to decide which state is changed. Time Set

esp_ble_mesh_state_change_time_status_t **time_status**

Time Status

esp_ble_mesh_state_change_time_zone_set_t **time_zone_set**

Time Zone Set

esp_ble_mesh_state_change_tai_utc_delta_set_t **tai_utc_delta_set**

TAI UTC Delta Set

esp_ble_mesh_state_change_time_role_set_t **time_role_set**

Time Role Set

esp_ble_mesh_state_change_scene_store_t **scene_store**

Scene Store

esp_ble_mesh_state_change_scene_recall_t **scene_recall**

Scene Recall

esp_ble_mesh_state_change_scene_delete_t **scene_delete**

Scene Delete

esp_ble_mesh_state_change_scheduler_act_set_t **scheduler_act_set**

Scheduler Action Set

union **esp_ble_mesh_time_scene_server_recv_get_msg_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received get message union.

Public Members

esp_ble_mesh_server_recv_scheduler_act_get_t **scheduler_act**

Scheduler Action Get

union **esp_ble_mesh_time_scene_server_recv_set_msg_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received set message union.

Public Members

esp_ble_mesh_server_recv_time_set_t **time**

Time Set

esp_ble_mesh_server_recv_time_zone_set_t **time_zone**

Time Zone Set

esp_ble_mesh_server_recv_tai_utc_delta_set_t **tai_utc_delta**

TAI-UTC Delta Set

esp_ble_mesh_server_recv_time_role_set_t **time_role**

Time Role Set

esp_ble_mesh_server_recv_scene_store_t **scene_store**

Scene Store/Scene Store Unack

esp_ble_mesh_server_recv_scene_recall_t **scene_recall**

Scene Recall/Scene Recall Unack

esp_ble_mesh_server_recv_scene_delete_t **scene_delete**

Scene Delete/Scene Delete Unack

esp_ble_mesh_server_recv_scheduler_act_set_t **scheduler_act**

Scheduler Action Set/Scheduler Action Set Unack

union **esp_ble_mesh_time_scene_server_recv_status_msg_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received status message union.

Public Members

esp_ble_mesh_server_recv_time_status_t **time_status**

Time Status

union **esp_ble_mesh_time_scene_server_cb_value_t**

#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model callback value union.

Public Members

esp_ble_mesh_time_scene_server_state_change_t **state_change**

ESP_BLE_MESH_TIME_SCENE_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_time_scene_server_recv_get_msg_t **get**

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_time_scene_server_recv_set_msg_t **set**

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_SET_MSG_EVT

esp_ble_mesh_time_scene_server_recv_status_msg_t **status**

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_STATUS_MSG_EVT

Structures

struct **esp_ble_mesh_time_set_t**

Bluetooth Mesh Time Scene Client Model Get and Set parameters structure.

Parameters of Time Set

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **sub_second**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_time_zone_set_t**

Parameters of Time Zone Set

Public Members

uint8_t **time_zone_offset_new**

Upcoming local time zone offset

uint8_t **tai_zone_change**[5]

TAI Seconds time of the upcoming Time Zone Offset change

struct **esp_ble_mesh_tai_utc_delta_set_t**

Parameters of TAI-UTC Delta Set

Public Members

uint16_t **tai_utc_delta_new**

Upcoming difference between TAI and UTC in seconds

uint16_t **padding**

Always 0b0. Other values are Prohibited.

uint8_t **tai_delta_change**[5]

TAI Seconds time of the upcoming TAI-UTC Delta change

struct **esp_ble_mesh_time_role_set_t**

Parameter of Time Role Set

Public Members

uint8_t **time_role**

The Time Role for the element

struct **esp_ble_mesh_scene_store_t**

Parameter of Scene Store

Public Members

uint16_t **scene_number**

The number of scenes to be stored

struct **esp_ble_mesh_scene_recall_t**

Parameters of Scene Recall

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **scene_number**

The number of scenes to be recalled

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_scene_delete_t**

Parameter of Scene Delete

Public Members

uint16_t **scene_number**

The number of scenes to be deleted

struct **esp_ble_mesh_scheduler_act_get_t**

Parameter of Scheduler Action Get

Public Members

uint8_t **index**

Index of the Schedule Register entry to get

struct **esp_ble_mesh_scheduler_act_set_t**

Parameters of Scheduler Action Set

Public Members

uint64_t **index**

Index of the Schedule Register entry to set

uint64_t **year**

Scheduled year for the action

uint64_t **month**

Scheduled month for the action

uint64_t **day**

Scheduled day of the month for the action

uint64_t **hour**

Scheduled hour for the action

uint64_t **minute**

Scheduled minute for the action

uint64_t **second**

Scheduled second for the action

uint64_t **day_of_week**

Schedule days of the week for the action

uint64_t **action**

Action to be performed at the scheduled time

uint64_t **trans_time**

Transition time for this action

uint16_t **scene_number**

Transition time for this action

struct **esp_ble_mesh_time_status_cb_t**

Bluetooth Mesh Time Scene Client Model Get and Set callback parameters structure.

Parameters of Time Status

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **sub_second**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_time_zone_status_cb_t**

Parameters of Time Zone Status

Public Members

uint8_t **time_zone_offset_curr**

Current local time zone offset

uint8_t **time_zone_offset_new**

Upcoming local time zone offset

uint8_t **tai_zone_change**[5]

TAI Seconds time of the upcoming Time Zone Offset change

struct **esp_ble_mesh_tai_utc_delta_status_cb_t**

Parameters of TAI-UTC Delta Status

Public Members

uint16_t **tai_utc_delta_curr**

Current difference between TAI and UTC in seconds

uint16_t **padding_1**

Always 0b0. Other values are Prohibited.

uint16_t **tai_utc_delta_new**

Upcoming difference between TAI and UTC in seconds

uint16_t **padding_2**

Always 0b0. Other values are Prohibited.

uint8_t **tai_delta_change**[5]

TAI Seconds time of the upcoming TAI-UTC Delta change

struct **esp_ble_mesh_time_role_status_cb_t**

Parameter of Time Role Status

Public Members

uint8_t **time_role**

The Time Role for the element

struct **esp_ble_mesh_scene_status_cb_t**

Parameters of Scene Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint8_t **status_code**

Status code of the last operation

uint16_t **current_scene**

Scene Number of the current scene

uint16_t **target_scene**

Scene Number of the target scene (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_scene_register_status_cb_t**

Parameters of Scene Register Status

Public Members

uint8_t **status_code**

Status code for the previous operation

uint16_t **current_scene**

Scene Number of the current scene

struct net_buf_simple ***scenes**

A list of scenes stored within an element

struct **esp_ble_mesh_scheduler_status_cb_t**

Parameter of Scheduler Status

Public Members

uint16_t **schedules**

Bit field indicating defined Actions in the Schedule Register

struct **esp_ble_mesh_scheduler_act_status_cb_t**

Parameters of Scheduler Action Status

Public Members

uint64_t **index**

Enumerates (selects) a Schedule Register entry

uint64_t **year**

Scheduled year for the action

uint64_t **month**

Scheduled month for the action

uint64_t **day**

Scheduled day of the month for the action

uint64_t **hour**

Scheduled hour for the action

uint64_t **minute**

Scheduled minute for the action

uint64_t **second**

Scheduled second for the action

uint64_t **day_of_week**

Schedule days of the week for the action

uint64_t **action**

Action to be performed at the scheduled time

uint64_t **trans_time**

Transition time for this action

uint16_t **scene_number**

Transition time for this action

struct **esp_ble_mesh_time_scene_client_cb_param_t**

Time Scene Client Model callback parameters

Public Members

int **error_code**

Appropriate error code

esp_ble_mesh_client_common_param_t ***params**

The client common parameters.

esp_ble_mesh_time_scene_client_status_cb_t **status_cb**

The scene status message callback values

struct **esp_ble_mesh_time_state_t**

Parameters of Time state

Public Members

uint8_t **tai_seconds**[5]

The value of the TAI Seconds state

uint8_t **subsecond**

The value of the Subsecond field

uint8_t **uncertainty**

The value of the Uncertainty field

uint8_t **time_zone_offset_curr**

The value of the Time Zone Offset Current field

uint8_t **time_zone_offset_new**

The value of the Time Zone Offset New state

uint8_t **tai_zone_change**[5]

The value of the TAI of Zone Chaneg field

uint16_t **time_authority**

The value of the Time Authority bit

uint16_t **tai_utc_delta_curr**

The value of the TAI-UTC Delta Current state

uint16_t **tai_utc_delta_new**

The value of the TAI-UTC Delta New state

uint8_t **tai_delta_change**[5]

The value of the TAI of Delta Change field

struct *esp_ble_mesh_time_state_t*::[anonymous] **time**

Parameters of the Time state

uint8_t **time_role**

The value of the Time Role state

struct **esp_ble_mesh_time_srv_t**

User data of Time Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Time Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_time_state_t ***state**

Parameters of the Time state

struct **esp_ble_mesh_time_setup_srv_t**

User data of Time Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Time Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_time_state_t ***state**

Parameters of the Time state

struct **esp_ble_mesh_scene_register_t**

a. Scene Store is an operation of storing values of a present state of an element.
b. The structure and meaning of the stored state is determined by a model. States to be stored are specified by each model.
c. The Scene Store operation shall persistently store all values of all states marked as Stored with Scene for all models present on all elements of a node.
d. If a model is extending another model, the extending model shall determine the Stored with Scene behavior of that model. Parameters of Scene Register state

Public Members

uint16_t **scene_number**

The value of the Scene Number

uint8_t **scene_type**

The value of the Scene Type

struct net_buf_simple ***scene_value**

Scene value may use a union to represent later, the union contains structures of all the model states which can be stored in a scene. The value of the Scene Value

struct **esp_ble_mesh_scenes_state_t**

Parameters of Scenes state.

Scenes serve as memory banks for storage of states (e.g., a power level or a light level/color). Values of states of an element can be stored as a scene and can be recalled later from the scene memory.

A scene is represented by a Scene Number, which is a 16-bit non-zero, mesh-wide value. (There can be a maximum of 65535 scenes in a mesh network.) The meaning of a scene, as well as the state storage container associated with it, are determined by a model.

The Scenes state change may start numerous parallel model transitions. In that case, each individual model handles the transition internally.

The scene transition is defined as a group of individual model transitions started by a Scene Recall operation. The scene transition is in progress when at least one transition from the group of individual model transitions is in progress.

Public Members

const uint16_t **scene_count**

The Scenes state' s scene count

esp_ble_mesh_scene_register_t ***scenes**

Parameters of the Scenes state

uint16_t **current_scene**

The Current Scene state is a 16-bit value that contains either the Scene Number of the currently active scene or a value of 0x0000 when no scene is active.

When a Scene Store operation or a Scene Recall operation completes with success, the Current Scene state value shall be to the Scene Number used during that operation.

When the Current Scene Number is deleted from a Scene Register state as a result of Scene Delete operation, the Current Scene state shall be set to 0x0000.

When any of the element's state that is marked as "Stored with Scene" has changed not as a result of a Scene Recall operation, the value of the Current Scene state shall be set to 0x0000.

When a scene transition is in progress, the value of the Current Scene state shall be set to 0x0000. The value of the Current Scene state

uint16_t **target_scene**

The Target Scene state is a 16-bit value that contains the target Scene Number when a scene transition is in progress.

When the scene transition is in progress and the target Scene Number is deleted from a Scene Register state as a result of Scene Delete operation, the Target Scene state shall be set to 0x0000.

When the scene transition is in progress and a new Scene Number is stored in the Scene Register as a result of Scene Store operation, the Target Scene state shall be set to the new Scene Number.

When the scene transition is not in progress, the value of the Target Scene state shall be set to 0x0000. The value of the Target Scene state

uint8_t **status_code**

The status code of the last scene operation

bool **in_progress**

Indicate if the scene transition is in progress

struct **esp_ble_mesh_scene_srv_t**

User data of Scene Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Scene Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_scenes_state_t ***state**

Parameters of the Scenes state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

struct **esp_ble_mesh_scene_setup_srv_t**

User data of Scene Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Scene Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_scenes_state_t ***state**

Parameters of the Scenes state

struct **esp_ble_mesh_schedule_register_t**

Parameters of Scheduler Register state

Public Members

bool **in_use**

Indicate if the registered schedule is in use

uint64_t **year**

The value of Scheduled year for the action

uint64_t **month**

The value of Scheduled month for the action

uint64_t **day**

The value of Scheduled day of the month for the action

uint64_t **hour**

The value of Scheduled hour for the action

uint64_t **minute**

The value of Scheduled minute for the action

uint64_t **second**

The value of Scheduled second for the action

uint64_t **day_of_week**

The value of Schedule days of the week for the action

uint64_t **action**

The value of Action to be performed at the scheduled time

uint64_t **trans_time**

The value of Transition time for this action

uint16_t **scene_number**

The value of Scene Number to be used for some actions

struct **esp_ble_mesh_scheduler_state_t**

Parameters of Scheduler state

Public Members

const uint8_t **schedule_count**

Scheduler count

esp_ble_mesh_schedule_register_t ***schedules**

Up to 16 scheduled entries

struct **esp_ble_mesh_scheduler_srv_t**

User data of Scheduler Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Scheduler Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_scheduler_state_t ***state**

Parameters of the Scheduler state

struct **esp_ble_mesh_scheduler_setup_srv_t**

User data of Scheduler Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Scheduler Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_scheduler_state_t ***state**

Parameters of the Scheduler state

struct **esp_ble_mesh_state_change_time_set_t**

Parameters of Time Set state change event

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **subsecond**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta_curr**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset_curr**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_state_change_time_status_t**

Parameters of Time Status state change event

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **subsecond**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta_curr**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset_curr**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_state_change_time_zone_set_t**

Parameters of Time Zone Set state change event

Public Members

uint8_t **time_zone_offset_new**

Upcoming local time zone offset

uint8_t **tai_zone_change**[5]

TAI Seconds time of the upcoming Time Zone Offset change

struct **esp_ble_mesh_state_change_tai_utc_delta_set_t**

Parameters of TAI UTC Delta Set state change event

Public Members

uint16_t **tai_utc_delta_new**

Upcoming difference between TAI and UTC in seconds

uint8_t **tai_delta_change**[5]

TAI Seconds time of the upcoming TAI-UTC Delta change

struct **esp_ble_mesh_state_change_time_role_set_t**

Parameter of Time Role Set state change event

Public Members

uint8_t **time_role**

The Time Role for the element

struct **esp_ble_mesh_state_change_scene_store_t**

Parameter of Scene Store state change event

Public Members

uint16_t **scene_number**

The number of scenes to be stored

struct **esp_ble_mesh_state_change_scene_recall_t**

Parameter of Scene Recall state change event

Public Members

uint16_t **scene_number**

The number of scenes to be recalled

struct **esp_ble_mesh_state_change_scene_delete_t**

Parameter of Scene Delete state change event

Public Members

uint16_t **scene_number**

The number of scenes to be deleted

struct **esp_ble_mesh_state_change_scheduler_act_set_t**

Parameter of Scheduler Action Set state change event

Public Members

uint64_t **index**

Index of the Schedule Register entry to set

uint64_t **year**

Scheduled year for the action

uint64_t **month**

Scheduled month for the action

uint64_t **day**

Scheduled day of the month for the action

uint64_t **hour**

Scheduled hour for the action

uint64_t **minute**

Scheduled minute for the action

uint64_t **second**

Scheduled second for the action

uint64_t **day_of_week**

Schedule days of the week for the action

uint64_t **action**

Action to be performed at the scheduled time

uint64_t **trans_time**

Transition time for this action

uint16_t **scene_number**

Scene number to be used for some actions

struct **esp_ble_mesh_server_recv_scheduler_act_get_t**

Context of the received Scheduler Action Get message

Public Members

uint8_t **index**

Index of the Schedule Register entry to get

struct **esp_ble_mesh_server_recv_time_set_t**

Context of the received Time Set message

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **subsecond**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_server_recv_time_zone_set_t**

Context of the received Time Zone Set message

Public Members

uint8_t **time_zone_offset_new**

Upcoming local time zone offset

uint8_t **tai_zone_change**[5]

TAI Seconds time of the upcoming Time Zone Offset change

struct **esp_ble_mesh_server_recv_tai_utc_delta_set_t**

Context of the received TAI UTC Delta Set message

Public Members

uint16_t **tai_utc_delta_new**

Upcoming difference between TAI and UTC in seconds

uint16_t **padding**

Always 0b0. Other values are Prohibited.

uint8_t **tai_delta_change**[5]

TAI Seconds time of the upcoming TAI-UTC Delta change

struct **esp_ble_mesh_server_recv_time_role_set_t**

Context of the received Time Role Set message

Public Members

uint8_t **time_role**

The Time Role for the element

struct **esp_ble_mesh_server_recv_scene_store_t**

Context of the received Scene Store message

Public Members

uint16_t **scene_number**

The number of scenes to be stored

struct **esp_ble_mesh_server_recv_scene_recall_t**

Context of the received Scene Recall message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **scene_number**

The number of scenes to be recalled

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_scene_delete_t**

Context of the received Scene Delete message

Public Members

uint16_t **scene_number**

The number of scenes to be deleted

struct **esp_ble_mesh_server_recv_scheduler_act_set_t**

Context of the received Scheduler Action Set message

Public Members

uint64_t **index**

Index of the Schedule Register entry to set

uint64_t **year**

Scheduled year for the action

uint64_t **month**

Scheduled month for the action

uint64_t **day**

Scheduled day of the month for the action

uint64_t **hour**

Scheduled hour for the action

uint64_t **minute**

Scheduled minute for the action

uint64_t **second**

Scheduled second for the action

uint64_t **day_of_week**

Schedule days of the week for the action

uint64_t **action**

Action to be performed at the scheduled time

uint64_t **trans_time**

Transition time for this action

uint16_t **scene_number**

Scene number to be used for some actions

struct **esp_ble_mesh_server_recv_time_status_t**

Context of the received Time Status message

Public Members

uint8_t **tai_seconds**[5]

The current TAI time in seconds

uint8_t **subsecond**

The sub-second time in units of 1/256 second

uint8_t **uncertainty**

The estimated uncertainty in 10-millisecond steps

uint16_t **time_authority**

0 = No Time Authority, 1 = Time Authority

uint16_t **tai_utc_delta**

Current difference between TAI and UTC in seconds

uint8_t **time_zone_offset**

The local time zone offset in 15-minute increments

struct **esp_ble_mesh_time_scene_server_cb_param_t**

Time Scene Server Model callback parameters

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Time and Scenes Server Models

esp_ble_mesh_msg_ctx_t **ctx**

Context of the received messages

esp_ble_mesh_time_scene_server_cb_value_t **value**

Value of the received Time and Scenes Messages

Macros

ESP_BLE_MESH_MODEL_TIME_CLI(cli_pub, cli_data)

Define a new Time Client Model.

Note: This API needs to be called for each element on which the application needs to have a Time Client Model.

Parameters

- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Time Client Model instance.

ESP_BLE_MESH_MODEL_SCENE_CLI(cli_pub, cli_data)

Define a new Scene Client Model.

Note: This API needs to be called for each element on which the application needs to have a Scene Client Model.

Parameters

- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Scene Client Model instance.

ESP_BLE_MESH_MODEL_SCHEDULER_CLI (cli_pub, cli_data)

Define a new Scheduler Client Model.

Note: This API needs to be called for each element on which the application needs to have a Scheduler Client Model.

Parameters

- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Scheduler Client Model instance.

ESP_BLE_MESH_MODEL_TIME_SRV (srv_pub, srv_data)

Time Scene Server Models related context.

Define a new Time Server Model.

Note: 1. The Time Server model is a root model. When this model is present on an Element, the corresponding Time Setup Server model shall also be present.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_time_srv_t*.

Returns New Time Server Model instance.

ESP_BLE_MESH_MODEL_TIME_SETUP_SRV (srv_data)

Define a new Time Setup Server Model.

Note: 1. The Time Setup Server model extends the Time Server model. Time is sensitive information that is propagated across a mesh network.

a. Only an authorized Time Client should be allowed to change the Time and Time Role states. A dedicated application key Bluetooth SIG Proprietary should be used on the Time Setup Server to restrict access to the server to only authorized Time Clients.
b. This model does not support subscribing nor publishing.

Parameters

- **srv_data** –Pointer to the unique struct *esp_ble_mesh_time_setup_srv_t*.

Returns New Time Setup Server Model instance.

ESP_BLE_MESH_MODEL_SCENE_SRV (srv_pub, srv_data)

Define a new Scene Server Model.

Note: 1. The Scene Server model is a root model. When this model is present on an Element, the corresponding Scene Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. The model may be present only on the Primary element of a node.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_scene_srv_t*.

Returns New Scene Server Model instance.

ESP_BLE_MESH_MODEL_SCENE_SETUP_SRV(srv_pub, srv_data)

Define a new Scene Setup Server Model.

Note: 1. The Scene Setup Server model extends the Scene Server model and the Generic Default Transition Time Server model.

a. This model shall support model subscription.
b. The model may be present only on the Primary element of a node.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_scene_setup_srv_t*.

Returns New Scene Setup Server Model instance.

ESP_BLE_MESH_MODEL_SCHEDULER_SRV(srv_pub, srv_data)

Define a new Scheduler Server Model.

Note: 1. The Scheduler Server model extends the Scene Server model. When this model is present on an Element, the corresponding Scheduler Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. The model may be present only on the Primary element of a node.
c. The model requires the Time Server model shall be present on the element.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_scheduler_srv_t*.

Returns New Scheduler Server Model instance.

ESP_BLE_MESH_MODEL_SCHEDULER_SETUP_SRV(srv_pub, srv_data)

Define a new Scheduler Setup Server Model.

Note: 1. The Scheduler Setup Server model extends the Scheduler Server and the Scene Setup Server models.

a. This model shall support model subscription.
b. The model may be present only on the Primary element of a node.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_scheduler_setup_srv_t*.

Returns New Scheduler Setup Server Model instance.

ESP_BLE_MESH_UNKNOWN_TAI_SECONDS

Unknown TAI Seconds

ESP_BLE_MESH_UNKNOWN_TAI_ZONE_CHANGE

Unknown TAI of Zone Change

ESP_BLE_MESH_UNKNOWN_TAI_DELTA_CHANGE

Unknown TAI of Delta Change

ESP_BLE_MESH_TAI_UTC_DELTA_MAX_VALUE

Maximum TAI-UTC Delta value

ESP_BLE_MESH_TAI_SECONDS_LEN

Length of TAI Seconds

ESP_BLE_MESH_TAI_OF_ZONE_CHANGE_LEN

Length of TAI of Zone Change

ESP_BLE_MESH_TAI_OF_DELTA_CHANGE_LEN

Length of TAI of Delta Change

ESP_BLE_MESH_INVALID_SCENE_NUMBER

Invalid Scene Number

ESP_BLE_MESH_SCENE_NUMBER_LEN

Length of the Scene Number

ESP_BLE_MESH_SCHEDULE_YEAR_ANY_YEAR

Any year of the Scheduled year

ESP_BLE_MESH_SCHEDULE_DAY_ANY_DAY

Any day of the Scheduled day

ESP_BLE_MESH_SCHEDULE_HOUR_ANY_HOUR

Any hour of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_HOUR_ONCE_A_DAY

Any hour of the Scheduled Day

ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_HOUR

Any minute of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_EVERY_15_MIN

Every 15 minutes of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_EVERY_20_MIN

Every 20 minutes of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_HOUR

Once of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_MIN

Any second of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC_EVERY_15_SEC

Every 15 seconds of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC_EVERY_20_SEC

Every 20 seconds of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_MIN

Once of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_ACT_TURN_OFF

Scheduled Action - Turn Off

ESP_BLE_MESH_SCHEDULE_ACT_TURN_ON

Scheduled Action - Turn On

ESP_BLE_MESH_SCHEDULE_ACT_SCENE_RECALL

Scheduled Action - Scene Recall

ESP_BLE_MESH_SCHEDULE_ACT_NO_ACTION

Scheduled Action - No Action

ESP_BLE_MESH_SCHEDULE_SCENE_NO_SCENE

Scheduled Scene - No Scene

ESP_BLE_MESH_SCHEDULE_ENTRY_MAX_INDEX

Maximum number of Scheduled entries

ESP_BLE_MESH_TIME_NONE

Time Role - None

ESP_BLE_MESH_TIME_AUTHORITY

Time Role - Mesh Time Authority

ESP_BLE_MESH_TIME_RELAY

Time Role - Mesh Time Relay

ESP_BLE_MESH_TIME_CLINET

Time Role - Mesh Time Client

ESP_BLE_MESH_SCENE_SUCCESS

Scene operation - Success

ESP_BLE_MESH_SCENE_REG_FULL

Scene operation - Scene Register Full

ESP_BLE_MESH_SCENE_NOT_FOUND

Scene operation - Scene Not Found

Type Definitions

typedef void (*`esp_ble_mesh_time_scene_client_cb_t`)(*esp_ble_mesh_time_scene_client_cb_event_t* event, *esp_ble_mesh_time_scene_client_cb_param_t* *param)

Bluetooth Mesh Time Scene Client Model function.

Time Scene Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (*`esp_ble_mesh_time_scene_server_cb_t`)(*esp_ble_mesh_time_scene_server_cb_event_t* event, *esp_ble_mesh_time_scene_server_cb_param_t* *param)

Bluetooth Mesh Time and Scenes Server Model function.

Time Scene Server Model callback function type

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum `esp_ble_mesh_time_scene_client_cb_event_t`

This enum value is the event of Time Scene Client Model

Values:

enumerator `ESP_BLE_MESH_TIME_SCENE_CLIENT_GET_STATE_EVT`

enumerator `ESP_BLE_MESH_TIME_SCENE_CLIENT_SET_STATE_EVT`

enumerator `ESP_BLE_MESH_TIME_SCENE_CLIENT_PUBLISH_EVT`

enumerator `ESP_BLE_MESH_TIME_SCENE_CLIENT_TIMEOUT_EVT`

enumerator `ESP_BLE_MESH_TIME_SCENE_CLIENT_EVT_MAX`

enum `esp_ble_mesh_time_scene_server_cb_event_t`

This enum value is the event of Time Scene Server Model

Values:

enumerator `ESP_BLE_MESH_TIME_SCENE_SERVER_STATE_CHANGE_EVT`

i. When get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, no event will be callback to the application layer when Time Scene Get messages are received.
ii. When set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, this event will be callback to the application layer when Time Scene Set/Set Unack messages are received.

enumerator `ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_GET_MSG_EVT`

When get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Time Scene Get messages are received.

enumerator `ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_SET_MSG_EVT`

When set_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Time Scene Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_STATUS_MSG_EVT**

When status_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when TIme Status message is received.

enumerator **ESP_BLE_MESH_TIME_SCENE_SERVER_EVT_MAX**

Lighting Client/Server Models

Header File

- components/bt/esp_ble_mesh/api/models/include/esp_ble_mesh_lighting_model_api.h

Functions

esp_err_t **esp_ble_mesh_register_light_client_callback**(*esp_ble_mesh_light_client_cb_t* callback)

Register BLE Mesh Light Client Model callback.

Parameters **callback** –**[in]** pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_light_client_get_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_light_client_get_state_t* *get_state)

Get the value of Light Server Model states using the Light Client Model get messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_light_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **get_state** –**[in]** Pointer of light get message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_light_client_set_state**(*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_light_client_set_state_t* *set_state)

Set the value of Light Server Model states using the Light Client Model set messages.

Note: If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_light_message_opcode_t in esp_ble_mesh_defs.h

---

Parameters
- **params** –**[in]** Pointer to BLE Mesh common client parameters.
- **set_state** –**[in]** Pointer of light set message value. Shall not be set to NULL.

Returns ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_register_lighting_server_callback**(*esp_ble_mesh_lighting_server_cb_t* callback)

Register BLE Mesh Lighting Server Model callback.

Parameters **callback** –**[in]** Pointer to the callback function.
Returns ESP_OK on success or error code otherwise.

Unions

union **esp_ble_mesh_light_client_get_state_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Client Model get message union.

Public Members

esp_ble_mesh_light_lc_property_get_t **lc_property_get**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET

union **esp_ble_mesh_light_client_set_state_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Client Model set message union.

Public Members

esp_ble_mesh_light_lightness_set_t **lightness_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK

esp_ble_mesh_light_lightness_linear_set_t **lightness_linear_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET_UNACK

esp_ble_mesh_light_lightness_default_set_t **lightness_default_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET_UNACK

esp_ble_mesh_light_lightness_range_set_t **lightness_range_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET_UNACK

esp_ble_mesh_light_ctl_set_t **ctl_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET_UNA

esp_ble_mesh_light_ctl_temperature_set_t **ctl_temperature_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET_UNACK

esp_ble_mesh_light_ctl_temperature_range_set_t **ctl_temperature_range_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET_UNACK

esp_ble_mesh_light_ctl_default_set_t **ctl_default_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET_UNACK

esp_ble_mesh_light_hsl_set_t **hsl_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET_UNA

esp_ble_mesh_light_hsl_hue_set_t **hsl_hue_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE

esp_ble_mesh_light_hsl_saturation_set_t **hsl_saturation_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET_UNACK

esp_ble_mesh_light_hsl_default_set_t **hsl_default_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET_UNACK

esp_ble_mesh_light_hsl_range_set_t **hsl_range_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET_UNACK

esp_ble_mesh_light_xyl_set_t **xyl_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET_UNA

esp_ble_mesh_light_xyl_default_set_t **xyl_default_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET_UNACK

esp_ble_mesh_light_xyl_range_set_t **xyl_range_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET_UNACK

esp_ble_mesh_light_lc_mode_set_t **lc_mode_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET_UNACK

esp_ble_mesh_light_lc_om_set_t **lc_om_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET

esp_ble_mesh_light_lc_light_onoff_set_t **lc_light_onoff_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET_UNACK

esp_ble_mesh_light_lc_property_set_t **lc_property_set**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK

union **esp_ble_mesh_light_client_status_cb_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Client Model received message union.

Public Members

esp_ble_mesh_light_lightness_status_cb_t **lightness_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS

esp_ble_mesh_light_lightness_linear_status_cb_t **lightness_linear_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_STATUS

esp_ble_mesh_light_lightness_last_status_cb_t **lightness_last_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_STATUS

esp_ble_mesh_light_lightness_default_status_cb_t **lightness_default_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS

esp_ble_mesh_light_lightness_range_status_cb_t **lightness_range_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_STATUS

esp_ble_mesh_light_ctl_status_cb_t **ctl_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_STATUS

esp_ble_mesh_light_ctl_temperature_status_cb_t **ctl_temperature_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_STATUS

esp_ble_mesh_light_ctl_temperature_range_status_cb_t **ctl_temperature_range_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_STATUS

esp_ble_mesh_light_ctl_default_status_cb_t **ctl_default_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_STATUS

esp_ble_mesh_light_hsl_status_cb_t **hsl_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_STATUS

esp_ble_mesh_light_hsl_target_status_cb_t **hsl_target_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_STATUS

esp_ble_mesh_light_hsl_hue_status_cb_t **hsl_hue_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_STATUS

esp_ble_mesh_light_hsl_saturation_status_cb_t **hsl_saturation_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_STATUS

esp_ble_mesh_light_hsl_default_status_cb_t **hsl_default_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_STATUS

esp_ble_mesh_light_hsl_range_status_cb_t **hsl_range_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_STATUS

esp_ble_mesh_light_xyl_status_cb_t **xyl_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_STATUS

esp_ble_mesh_light_xyl_target_status_cb_t **xyl_target_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_STATUS

esp_ble_mesh_light_xyl_default_status_cb_t **xyl_default_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_STATUS

esp_ble_mesh_light_xyl_range_status_cb_t **xyl_range_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_STATUS

esp_ble_mesh_light_lc_mode_status_cb_t **lc_mode_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_STATUS

esp_ble_mesh_light_lc_om_status_cb_t **lc_om_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS

esp_ble_mesh_light_lc_light_onoff_status_cb_t **lc_light_onoff_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_STATUS

esp_ble_mesh_light_lc_property_status_cb_t **lc_property_status**

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS

union **esp_ble_mesh_lighting_server_state_change_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model state change value union.

Public Members

esp_ble_mesh_state_change_light_lightness_set_t **lightness_set**

The recv_op in ctx can be used to decide which state is changed. Light Lightness Set

esp_ble_mesh_state_change_light_lightness_linear_set_t **lightness_linear_set**

Light Lightness Linear Set

esp_ble_mesh_state_change_light_lightness_default_set_t **lightness_default_set**

Light Lightness Default Set

esp_ble_mesh_state_change_light_lightness_range_set_t **lightness_range_set**

Light Lightness Range Set

esp_ble_mesh_state_change_light_ctl_set_t **ctl_set**

Light CTL Set

esp_ble_mesh_state_change_light_ctl_temperature_set_t **ctl_temp_set**

Light CTL Temperature Set

esp_ble_mesh_state_change_light_ctl_temperature_range_set_t **ctl_temp_range_set**

Light CTL Temperature Range Set

esp_ble_mesh_state_change_light_ctl_default_set_t **ctl_default_set**

Light CTL Default Set

esp_ble_mesh_state_change_light_hsl_set_t **hsl_set**
Light HSL Set

esp_ble_mesh_state_change_light_hsl_hue_set_t **hsl_hue_set**
Light HSL Hue Set

esp_ble_mesh_state_change_light_hsl_saturation_set_t **hsl_saturation_set**
Light HSL Saturation Set

esp_ble_mesh_state_change_light_hsl_default_set_t **hsl_default_set**
Light HSL Default Set

esp_ble_mesh_state_change_light_hsl_range_set_t **hsl_range_set**
Light HSL Range Set

esp_ble_mesh_state_change_light_xyl_set_t **xyl_set**
Light xyL Set

esp_ble_mesh_state_change_light_xyl_default_set_t **xyl_default_set**
Light xyL Default Set

esp_ble_mesh_state_change_light_xyl_range_set_t **xyl_range_set**
Light xyL Range Set

esp_ble_mesh_state_change_light_lc_mode_set_t **lc_mode_set**
Light LC Mode Set

esp_ble_mesh_state_change_light_lc_om_set_t **lc_om_set**
Light LC Occupancy Mode Set

esp_ble_mesh_state_change_light_lc_light_onoff_set_t **lc_light_onoff_set**
Light LC Light OnOff Set

esp_ble_mesh_state_change_light_lc_property_set_t **lc_property_set**
Light LC Property Set

esp_ble_mesh_state_change_sensor_status_t **sensor_status**
Sensor Status

union **esp_ble_mesh_lighting_server_recv_get_msg_t**
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received get message union.

Public Members

esp_ble_mesh_server_recv_light_lc_property_get_t **lc_property**
Light LC Property Get

union **esp_ble_mesh_lighting_server_recv_set_msg_t**
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received set message union.

Public Members

esp_ble_mesh_server_recv_light_lightness_set_t **lightness**

Light Lightness Set/Light Lightness Set Unack

esp_ble_mesh_server_recv_light_lightness_linear_set_t **lightness_linear**

Light Lightness Linear Set/Light Lightness Linear Set Unack

esp_ble_mesh_server_recv_light_lightness_default_set_t **lightness_default**

Light Lightness Default Set/Light Lightness Default Set Unack

esp_ble_mesh_server_recv_light_lightness_range_set_t **lightness_range**

Light Lightness Range Set/Light Lightness Range Set Unack

esp_ble_mesh_server_recv_light_ctl_set_t **ctl**

Light CTL Set/Light CTL Set Unack

esp_ble_mesh_server_recv_light_ctl_temperature_set_t **ctl_temp**

Light CTL Temperature Set/Light CTL Temperature Set Unack

esp_ble_mesh_server_recv_light_ctl_temperature_range_set_t **ctl_temp_range**

Light CTL Temperature Range Set/Light CTL Temperature Range Set Unack

esp_ble_mesh_server_recv_light_ctl_default_set_t **ctl_default**

Light CTL Default Set/Light CTL Default Set Unack

esp_ble_mesh_server_recv_light_hsl_set_t **hsl**

Light HSL Set/Light HSL Set Unack

esp_ble_mesh_server_recv_light_hsl_hue_set_t **hsl_hue**

Light HSL Hue Set/Light HSL Hue Set Unack

esp_ble_mesh_server_recv_light_hsl_saturation_set_t **hsl_saturation**

Light HSL Saturation Set/Light HSL Saturation Set Unack

esp_ble_mesh_server_recv_light_hsl_default_set_t **hsl_default**

Light HSL Default Set/Light HSL Default Set Unack

esp_ble_mesh_server_recv_light_hsl_range_set_t **hsl_range**

Light HSL Range Set/Light HSL Range Set Unack

esp_ble_mesh_server_recv_light_xyl_set_t **xyl**

Light xyL Set/Light xyL Set Unack

esp_ble_mesh_server_recv_light_xyl_default_set_t **xyl_default**

Light xyL Default Set/Light xyL Default Set Unack

esp_ble_mesh_server_recv_light_xyl_range_set_t **xyl_range**

Light xyL Range Set/Light xyL Range Set Unack

esp_ble_mesh_server_recv_light_lc_mode_set_t **lc_mode**

Light LC Mode Set/Light LC Mode Set Unack

esp_ble_mesh_server_recv_light_lc_om_set_t **lc_om**

Light LC OM Set/Light LC OM Set Unack

esp_ble_mesh_server_recv_light_lc_light_onoff_set_t **lc_light_onoff**

Light LC Light OnOff Set/Light LC Light OnOff Set Unack

esp_ble_mesh_server_recv_light_lc_property_set_t **lc_property**

Light LC Property Set/Light LC Property Set Unack

union **esp_ble_mesh_lighting_server_recv_status_msg_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received status message union.

Public Members

esp_ble_mesh_server_recv_sensor_status_t **sensor_status**

Sensor Status

union **esp_ble_mesh_lighting_server_cb_value_t**

#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model callback value union.

Public Members

esp_ble_mesh_lighting_server_state_change_t **state_change**

ESP_BLE_MESH_LIGHTING_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_lighting_server_recv_get_msg_t **get**

ESP_BLE_MESH_LIGHTING_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_lighting_server_recv_set_msg_t **set**

ESP_BLE_MESH_LIGHTING_SERVER_RECV_SET_MSG_EVT

esp_ble_mesh_lighting_server_recv_status_msg_t **status**

ESP_BLE_MESH_LIGHTING_SERVER_RECV_STATUS_MSG_EVT

Structures

struct **esp_ble_mesh_light_lightness_set_t**

Bluetooth Mesh Light Lightness Client Model Get and Set parameters structure.

Parameters of Light Lightness Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light lightness actual state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_lightness_linear_set_t**

Parameters of Light Lightness Linear Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light lightness linear state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_lightness_default_set_t**

Parameter of Light Lightness Default Set

Public Members

uint16_t **lightness**

The value of the Light Lightness Default state

struct **esp_ble_mesh_light_lightness_range_set_t**

Parameters of Light Lightness Range Set

Public Members

uint16_t **range_min**

Value of range min field of light lightness range state

uint16_t **range_max**

Value of range max field of light lightness range state

struct **esp_ble_mesh_light_ctl_set_t**

Parameters of Light CTL Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **ctl_lightness**

Target value of light ctl lightness state

uint16_t **ctl_temperatrue**

Target value of light ctl temperature state

int16_t **ctl_delta_uv**

Target value of light ctl delta UV state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_ctl_temperature_set_t**

Parameters of Light CTL Temperature Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **ctl_temperatrue**

Target value of light ctl temperature state

int16_t **ctl_delta_uv**

Target value of light ctl delta UV state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_ctl_temperature_range_set_t**

Parameters of Light CTL Temperature Range Set

Public Members

uint16_t **range_min**

Value of temperature range min field of light ctl temperature range state

uint16_t **range_max**

Value of temperature range max field of light ctl temperature range state

struct **esp_ble_mesh_light_ctl_default_set_t**

Parameters of Light CTL Default Set

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **temperature**

Value of light temperature default state

int16_t **delta_uv**

Value of light delta UV default state

struct **esp_ble_mesh_light_hsl_set_t**

Parameters of Light HSL Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **hsl_lightness**

Target value of light hsl lightness state

uint16_t **hsl_hue**

Target value of light hsl hue state

uint16_t **hsl_saturation**

Target value of light hsl saturation state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_hsl_hue_set_t**

Parameters of Light HSL Hue Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **hue**

Target value of light hsl hue state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_hsl_saturation_set_t**

Parameters of Light HSL Saturation Set

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **saturation**

Target value of light hsl hue state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_hsl_default_set_t**

Parameters of Light HSL Default Set

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **hue**

Value of light hue default state

uint16_t **saturation**

Value of light saturation default state

struct **esp_ble_mesh_light_hsl_range_set_t**

Parameters of Light HSL Range Set

Public Members

uint16_t **hue_range_min**

Value of hue range min field of light hsl hue range state

uint16_t **hue_range_max**

Value of hue range max field of light hsl hue range state

uint16_t **saturation_range_min**

Value of saturation range min field of light hsl saturation range state

uint16_t **saturation_range_max**

Value of saturation range max field of light hsl saturation range state

struct **esp_ble_mesh_light_xyl_set_t**

Parameters of Light xyL Set

Public Members

bool **op_en**

Indicate whether optional parameters included

uint16_t **xyl_lightness**

The target value of the Light xyL Lightness state

uint16_t **xyl_x**

The target value of the Light xyL x state

uint16_t **xyl_y**

The target value of the Light xyL y state

uint8_t **tid**

Transaction Identifier

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_xyl_default_set_t**

Parameters of Light xyL Default Set

Public Members

uint16_t **lightness**

The value of the Light Lightness Default state

uint16_t **xyl_x**

The value of the Light xyL x Default state

uint16_t **xyl_y**

The value of the Light xyL y Default state

struct **esp_ble_mesh_light_xyl_range_set_t**

Parameters of Light xyL Range Set

Public Members

uint16_t **xyl_x_range_min**

The value of the xyL x Range Min field of the Light xyL x Range state

uint16_t **xyl_x_range_max**

The value of the xyL x Range Max field of the Light xyL x Range state

uint16_t **xyl_y_range_min**

The value of the xyL y Range Min field of the Light xyL y Range state

uint16_t **xyl_y_range_max**

The value of the xyL y Range Max field of the Light xyL y Range state

struct **esp_ble_mesh_light_lc_mode_set_t**

Parameter of Light LC Mode Set

Public Members

uint8_t **mode**

The target value of the Light LC Mode state

struct **esp_ble_mesh_light_lc_om_set_t**

Parameter of Light LC OM Set

Public Members

uint8_t **mode**

The target value of the Light LC Occupancy Mode state

struct **esp_ble_mesh_light_lc_light_onoff_set_t**

Parameters of Light LC Light OnOff Set

Public Members

bool **op_en**

Indicate whether optional parameters included

uint8_t **light_onoff**

The target value of the Light LC Light OnOff state

uint8_t **tid**

Transaction Identifier

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_light_lc_property_get_t**

Parameter of Light LC Property Get

Public Members

uint16_t **property_id**

Property ID identifying a Light LC Property

struct **esp_ble_mesh_light_lc_property_set_t**

Parameters of Light LC Property Set

Public Members

uint16_t **property_id**

Property ID identifying a Light LC Property

struct net_buf_simple ***property_value**

Raw value for the Light LC Property

struct **esp_ble_mesh_light_lightness_status_cb_t**

Bluetooth Mesh Light Lightness Client Model Get and Set callback parameters structure.

Parameters of Light Lightness Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_lightness**

Current value of light lightness actual state

uint16_t **target_lightness**

Target value of light lightness actual state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_lightness_linear_status_cb_t**

Parameters of Light Lightness Linear Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_lightness**

Current value of light lightness linear state

uint16_t **target_lightness**

Target value of light lightness linear state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_lightness_last_status_cb_t**

Parameter of Light Lightness Last Status

Public Members

uint16_t **lightness**

The value of the Light Lightness Last state

struct **esp_ble_mesh_light_lightness_default_status_cb_t**

Parameter of Light Lightness Default Status

Public Members

uint16_t **lightness**

The value of the Light Lightness default State

struct **esp_ble_mesh_light_lightness_range_status_cb_t**

Parameters of Light Lightness Range Status

Public Members

uint8_t **status_code**

Status Code for the request message

uint16_t **range_min**

Value of range min field of light lightness range state

uint16_t **range_max**

Value of range max field of light lightness range state

struct **esp_ble_mesh_light_ctl_status_cb_t**

Parameters of Light CTL Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_ctl_lightness**

Current value of light ctl lightness state

uint16_t **present_ctl_temperature**

Current value of light ctl temperature state

uint16_t **target_ctl_lightness**

Target value of light ctl lightness state (optional)

uint16_t **target_ctl_temperature**

Target value of light ctl temperature state (C.1)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_ctl_temperature_status_cb_t**

Parameters of Light CTL Temperature Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_ctl_temperature**

Current value of light ctl temperature state

uint16_t **present_ctl_delta_uv**

Current value of light ctl delta UV state

uint16_t **target_ctl_temperature**

Target value of light ctl temperature state (optional)

uint16_t **target_ctl_delta_uv**

Target value of light ctl delta UV state (C.1)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_ctl_temperature_range_status_cb_t**

Parameters of Light CTL Temperature Range Status

Public Members

uint8_t **status_code**

Status code for the request message

uint16_t **range_min**

Value of temperature range min field of light ctl temperature range state

uint16_t **range_max**

Value of temperature range max field of light ctl temperature range state

struct **esp_ble_mesh_light_ctl_default_status_cb_t**

Parameters of Light CTL Default Status

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **temperature**

Value of light temperature default state

int16_t **delta_uv**

Value of light delta UV default state

struct **esp_ble_mesh_light_hsl_status_cb_t**

Parameters of Light HSL Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **hsl_lightness**

Current value of light hsl lightness state

uint16_t **hsl_hue**

Current value of light hsl hue state

uint16_t **hsl_saturation**

Current value of light hsl saturation state

uint8_t **remain_time**

Time to complete state transition (optional)

struct **esp_ble_mesh_light_hsl_target_status_cb_t**

Parameters of Light HSL Target Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **hsl_lightness_target**

Target value of light hsl lightness state

uint16_t **hsl_hue_target**

Target value of light hsl hue state

uint16_t **hsl_saturation_target**

Target value of light hsl saturation state

uint8_t **remain_time**

Time to complete state transition (optional)

struct **esp_ble_mesh_light_hsl_hue_status_cb_t**

Parameters of Light HSL Hue Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_hue**

Current value of light hsl hue state

uint16_t **target_hue**

Target value of light hsl hue state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_hsl_saturation_status_cb_t**

Parameters of Light HSL Saturation Status

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **present_saturation**

Current value of light hsl saturation state

uint16_t **target_saturation**

Target value of light hsl saturation state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_hsl_default_status_cb_t**

Parameters of Light HSL Default Status

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **hue**

Value of light hue default state

uint16_t **saturation**

Value of light saturation default state

struct **esp_ble_mesh_light_hsl_range_status_cb_t**

Parameters of Light HSL Range Status

Public Members

uint8_t **status_code**

Status code for the request message

uint16_t **hue_range_min**

Value of hue range min field of light hsl hue range state

uint16_t **hue_range_max**

Value of hue range max field of light hsl hue range state

uint16_t **saturation_range_min**

Value of saturation range min field of light hsl saturation range state

uint16_t **saturation_range_max**

Value of saturation range max field of light hsl saturation range state

struct **esp_ble_mesh_light_xyl_status_cb_t**

Parameters of Light xyL Status

Public Members

bool **op_en**

Indicate whether optional parameters included

uint16_t **xyl_lightness**

The present value of the Light xyL Lightness state

uint16_t **xyl_x**

The present value of the Light xyL x state

uint16_t **xyl_y**

The present value of the Light xyL y state

uint8_t **remain_time**

Time to complete state transition (optional)

struct **esp_ble_mesh_light_xyl_target_status_cb_t**

Parameters of Light xyL Target Status

Public Members

bool **op_en**

Indicate whether optional parameters included

uint16_t **target_xyl_lightness**

The target value of the Light xyL Lightness state

uint16_t **target_xyl_x**

The target value of the Light xyL x state

uint16_t **target_xyl_y**

The target value of the Light xyL y state

uint8_t **remain_time**

Time to complete state transition (optional)

struct **esp_ble_mesh_light_xyl_default_status_cb_t**

Parameters of Light xyL Default Status

Public Members

uint16_t **lightness**

The value of the Light Lightness Default state

uint16_t **xyl_x**

The value of the Light xyL x Default state

uint16_t **xyl_y**

The value of the Light xyL y Default state

struct **esp_ble_mesh_light_xyl_range_status_cb_t**

Parameters of Light xyL Range Status

Public Members

uint8_t **status_code**

Status Code for the requesting message

uint16_t **xyl_x_range_min**

The value of the xyL x Range Min field of the Light xyL x Range state

uint16_t **xyl_x_range_max**

The value of the xyL x Range Max field of the Light xyL x Range state

uint16_t **xyl_y_range_min**

The value of the xyL y Range Min field of the Light xyL y Range state

uint16_t **xyl_y_range_max**

The value of the xyL y Range Max field of the Light xyL y Range state

struct **esp_ble_mesh_light_lc_mode_status_cb_t**

Parameter of Light LC Mode Status

Public Members

uint8_t **mode**

The present value of the Light LC Mode state

struct **esp_ble_mesh_light_lc_om_status_cb_t**

Parameter of Light LC OM Status

Public Members

uint8_t **mode**

The present value of the Light LC Occupancy Mode state

struct **esp_ble_mesh_light_lc_light_onoff_status_cb_t**

Parameters of Light LC Light OnOff Status

Public Members

bool **op_en**

Indicate whether optional parameters included

uint8_t **present_light_onoff**

The present value of the Light LC Light OnOff state

uint8_t **target_light_onoff**

The target value of the Light LC Light OnOff state (Optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct **esp_ble_mesh_light_lc_property_status_cb_t**

Parameters of Light LC Property Status

Public Members

uint16_t **property_id**

Property ID identifying a Light LC Property

struct net_buf_simple ***property_value**

Raw value for the Light LC Property

struct **esp_ble_mesh_light_client_cb_param_t**

Lighting Client Model callback parameters

Public Members

int **error_code**

Appropriate error code

esp_ble_mesh_client_common_param_t ***params**

The client common parameters.

esp_ble_mesh_light_client_status_cb_t **status_cb**

The light status message callback values

struct **esp_ble_mesh_light_lightness_state_t**

Parameters of Light Lightness state

Public Members

uint16_t **lightness_linear**

The present value of Light Lightness Linear state

uint16_t **target_lightness_linear**

The target value of Light Lightness Linear state

uint16_t **lightness_actual**

The present value of Light Lightness Actual state

uint16_t **target_lightness_actual**

The target value of Light Lightness Actual state

uint16_t **lightness_last**

The value of Light Lightness Last state

uint16_t **lightness_default**

The value of Light Lightness Default state

uint8_t **status_code**

The status code of setting Light Lightness Range state

uint16_t **lightness_range_min**

The minimum value of Light Lightness Range state

uint16_t **lightness_range_max**

The maximum value of Light Lightness Range state

struct **esp_ble_mesh_light_lightness_srv_t**

User data of Light Lightness Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting Lightness Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_lightness_state_t ***state**

Parameters of the Light Lightness state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **actual_transition**

Parameters of state transition

esp_ble_mesh_state_transition_t **linear_transition**

Parameters of state transition

int32_t **tt_delta_lightness_actual**

Delta change value of lightness actual state transition

int32_t **tt_delta_lightness_linear**

Delta change value of lightness linear state transition

struct **esp_ble_mesh_light_lightness_setup_srv_t**

User data of Light Lightness Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting Lightness Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_lightness_state_t ***state**

Parameters of the Light Lightness state

struct **esp_ble_mesh_light_ctl_state_t**

Parameters of Light CTL state

Public Members

uint16_t **lightness**

The present value of Light CTL Lightness state

uint16_t **target_lightness**

The target value of Light CTL Lightness state

uint16_t **temperature**

The present value of Light CTL Temperature state

uint16_t **target_temperature**

The target value of Light CTL Temperature state

int16_t **delta_uv**

The present value of Light CTL Delta UV state

int16_t **target_delta_uv**

The target value of Light CTL Delta UV state

uint8_t **status_code**

The statue code of setting Light CTL Temperature Range state

uint16_t **temperature_range_min**

The minimum value of Light CTL Temperature Range state

uint16_t **temperature_range_max**

The maximum value of Light CTL Temperature Range state

uint16_t **lightness_default**

The value of Light Lightness Default state

uint16_t **temperature_default**

The value of Light CTL Temperature Default state

int16_t **delta_uv_default**

The value of Light CTL Delta UV Default state

struct **esp_ble_mesh_light_ctl_srv_t**

User data of Light CTL Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_lightness**

Delta change value of lightness state transition

int32_t **tt_delta_temperature**

Delta change value of temperature state transition

int32_t **tt_delta_delta_uv**

Delta change value of delta uv state transition

struct **esp_ble_mesh_light_ctl_setup_srv_t**

User data of Light CTL Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

struct **esp_ble_mesh_light_ctl_temp_srv_t**

User data of Light CTL Temperature Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Temperature Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_temperature**

Delta change value of temperature state transition

int32_t **tt_delta_delta_uv**

Delta change value of delta uv state transition

struct **esp_ble_mesh_light_hsl_state_t**

Parameters of Light HSL state

Public Members

uint16_t **lightness**

The present value of Light HSL Lightness state

uint16_t **target_lightness**

The target value of Light HSL Lightness state

uint16_t **hue**

The present value of Light HSL Hue state

uint16_t **target_hue**

The target value of Light HSL Hue state

uint16_t **saturation**

The present value of Light HSL Saturation state

uint16_t **target_saturation**

The target value of Light HSL Saturation state

uint16_t **lightness_default**

The value of Light Lightness Default state

uint16_t **hue_default**

The value of Light HSL Hue Default state

uint16_t **saturation_default**

The value of Light HSL Saturation Default state

uint8_t **status_code**

The status code of setting Light HSL Hue & Saturation Range state

uint16_t **hue_range_min**

The minimum value of Light HSL Hue Range state

uint16_t **hue_range_max**

The maximum value of Light HSL Hue Range state

uint16_t **saturation_range_min**

The minimum value of Light HSL Saturation state

uint16_t **saturation_range_max**

The maximum value of Light HSL Saturation state

struct **esp_ble_mesh_light_hsl_srv_t**

User data of Light HSL Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting HSL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_hsl_state_t ***state**

Parameters of the Light HSL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_lightness**

Delta change value of lightness state transition

int32_t **tt_delta_hue**

Delta change value of hue state transition

int32_t **tt_delta_saturation**

Delta change value of saturation state transition

struct **esp_ble_mesh_light_hsl_setup_srv_t**

User data of Light HSL Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting HSL Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_hsl_state_t ***state**

Parameters of the Light HSL state

struct **esp_ble_mesh_light_hsl_hue_srv_t**

User data of Light HSL Hue Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting HSL Hue Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_hsl_state_t ***state**

Parameters of the Light HSL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_hue**

Delta change value of hue state transition

struct **esp_ble_mesh_light_hsl_sat_srv_t**

User data of Light HSL Saturation Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting HSL Saturation Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_hsl_state_t ***state**

Parameters of the Light HSL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_saturation**

Delta change value of saturation state transition

struct **esp_ble_mesh_light_xyl_state_t**

Parameters of Light xyL state

Public Members

uint16_t **lightness**

The present value of Light xyL Lightness state

uint16_t **target_lightness**

The target value of Light xyL Lightness state

uint16_t **x**

The present value of Light xyL x state

uint16_t **target_x**

The target value of Light xyL x state

uint16_t **y**

The present value of Light xyL y state

uint16_t **target_y**

The target value of Light xyL y state

uint16_t **lightness_default**

The value of Light Lightness Default state

uint16_t **x_default**

The value of Light xyL x Default state

uint16_t **y_default**

The value of Light xyL y Default state

uint8_t **status_code**

The status code of setting Light xyL x & y Range state

uint16_t **x_range_min**

The minimum value of Light xyL x Range state

uint16_t **x_range_max**

The maximum value of Light xyL x Range state

uint16_t **y_range_min**

The minimum value of Light xyL y Range state

uint16_t **y_range_max**

The maximum value of Light xyL y Range state

struct **esp_ble_mesh_light_xyl_srv_t**

User data of Light xyL Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting xyL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_xyl_state_t ***state**

Parameters of the Light xyL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_lightness**

Delta change value of lightness state transition

int32_t **tt_delta_x**

Delta change value of x state transition

int32_t **tt_delta_y**

Delta change value of y state transition

struct **esp_ble_mesh_light_xyl_setup_srv_t**

User data of Light xyL Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting xyL Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_xyl_state_t ***state**

Parameters of the Light xyL state

struct **esp_ble_mesh_light_lc_state_t**

Parameters of Light LC states

Public Members

uint32_t **mode**

0b0 The controller is turned off.

- The binding with the Light Lightness state is disabled. 0b1 The controller is turned on.
- The binding with the Light Lightness state is enabled. The value of Light LC Mode state

uint32_t **occupancy_mode**

The value of Light LC Occupancy Mode state

uint32_t **light_onoff**

The present value of Light LC Light OnOff state

uint32_t **target_light_onoff**

The target value of Light LC Light OnOff state

uint32_t **occupancy**

The value of Light LC Occupancy state

uint32_t **ambient_luxlevel**

The value of Light LC Ambient LuxLevel state

uint16_t **linear_output**

i. Light LC Linear Output = max((Lightness Out)^2/65535, Regulator Output)
ii. If the Light LC Mode state is set to 0b1, the binding is enabled and upon a change of the Light LC Linear Output state, the following operation shall be performed: Light Lightness Linear = Light LC Linear Output
iii. If the Light LC Mode state is set to 0b0, the binding is disabled (i.e., upon a change of the Light LC Linear Output state, no operation on the Light Lightness Linear state is performed). The value of Light LC Linear Output state

struct **esp_ble_mesh_light_lc_property_state_t**

Parameters of Light Property states. The Light LC Property states are read / write states that determine the configuration of a Light Lightness Controller. Each state is represented by a device property and is controlled by Light LC Property messages.

Public Members

uint32_t **time_occupancy_delay**

A timing state that determines the delay for changing the Light LC Occupancy state upon receiving a Sensor Status message from an occupancy sensor. The value of Light LC Time Occupancy Delay state

uint32_t **time_fade_on**

A timing state that determines the time the controlled lights fade to the level determined by the Light LC Lightness On state. The value of Light LC Time Fade On state

uint32_t **time_run_on**

A timing state that determines the time the controlled lights stay at the level determined by the Light LC Lightness On state. The value of Light LC Time Run On state

uint32_t **time_fade**

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness On state to the level determined by the Light Lightness Prolong state. The value of Light LC Time Fade state

uint32_t **time_prolong**

A timing state that determines the time the controlled lights stay at the level determined by the Light LC Lightness Prolong state. The value of Light LC Time Prolong state

uint32_t **time_fade_standby_auto**

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness Prolong state to the level determined by the Light LC Lightness Standby state when the transition is automatic. The value of Light LC Time Fade Standby Auto state

uint32_t **time_fade_standby_manual**

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness Prolong state to the level determined by the Light LC Lightness Standby state when the transition is triggered by a change in the Light LC Light OnOff state. The value of Light LC Time Fade Standby Manual state

uint16_t **lightness_on**

A lightness state that determines the perceptive light lightness at the Occupancy and Run internal controller states. The value of Light LC Lightness On state

uint16_t **lightness_prolong**

A lightness state that determines the light lightness at the Prolong internal controller state. The value of Light LC Lightness Prolong state

uint16_t **lightness_standby**

A lightness state that determines the light lightness at the Standby internal controller state. The value of Light LC Lightness Standby state

uint16_t **ambient_luxlevel_on**

A uint16 state representing the Ambient LuxLevel level that determines if the controller transitions from the Light Control Standby state. The value of Light LC Ambient LuxLevel On state

uint16_t **ambient_luxlevel_prolong**

A uint16 state representing the required Ambient LuxLevel level in the Prolong state. The value of Light LC Ambient LuxLevel Prolong state

uint16_t **ambient_luxlevel_standby**

A uint16 state representing the required Ambient LuxLevel level in the Standby state. The value of Light LC Ambient LuxLevel Standby state

float **regulator_kiu**

A float32 state representing the integral coefficient that determines the integral part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is less than LuxLevel Out. Valid range: 0.0 ~ 1000.0. The default value is 250.0. The value of Light LC Regulator Kiu state

float **regulator_kid**

A float32 state representing the integral coefficient that determines the integral part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is greater than or equal to the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 25.0. The value of Light LC Regulator Kid state

float **regulator_kpu**

A float32 state representing the proportional coefficient that determines the proportional part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is less than the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 80.0. The value of Light LC Regulator Kpu state

float **regulator_kpd**

A float32 state representing the proportional coefficient that determines the proportional part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is greater than or equal to the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 80.0. The value of Light LC Regulator Kpd state

int8_t **regulator_accuracy**

A int8 state representing the percentage accuracy of the Light LC PI Feedback Regulator. Valid range: 0.0 ~ 100.0. The default value is 2.0. The value of Light LC Regulator Accuracy state

uint32_t **set_occupancy_to_1_delay**

If the message Raw field contains a Raw Value for the Time Since Motion Sensed device property, which represents a value less than or equal to the value of the Light LC Occupancy Delay state, it shall delay setting the Light LC Occupancy state to 0b1 by the difference between the value of the Light LC Occupancy Delay state and the received Time Since Motion value. The value of the difference between value of the Light LC Occupancy Delay state and the received Time Since Motion value

struct **esp_ble_mesh_light_lc_state_machine_t**

Parameters of Light LC state machine

Public Members

uint8_t **fade_on**

The value of transition time of Light LC Time Fade On

uint8_t **fade**

The value of transition time of Light LC Time Fade

uint8_t **fade_standby_auto**

The value of transition time of Light LC Time Fade Standby Auto

uint8_t **fade_standby_manual**

The value of transition time of Light LC Time Fade Standby Manual

struct *esp_ble_mesh_light_lc_state_machine_t*::[anonymous] **trans_time**

The Fade On, Fade, Fade Standby Auto, and Fade Standby Manual states are transition states that define the transition of the Lightness Out and LuxLevel Out states. This transition can be started as a result of the Light LC State Machine change or as a result of receiving the Light LC Light OnOff Set or Light LC Light Set Unacknowledged message. The value of transition time

esp_ble_mesh_lc_state_t **state**

The value of Light LC state machine state

struct k_delayed_work **timer**

Timer of Light LC state machine

struct **esp_ble_mesh_light_control_t**

Parameters of Light Lightness controller

Public Members

esp_ble_mesh_light_lc_state_t **state**

Parameters of Light LC state

esp_ble_mesh_light_lc_property_state_t **prop_state**

Parameters of Light LC Property state

esp_ble_mesh_light_lc_state_machine_t **state_machine**

Parameters of Light LC state machine

struct **esp_ble_mesh_light_lc_srv_t**

User data of Light LC Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting LC Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_control_t ***lc**

Parameters of the Light controller

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

struct **esp_ble_mesh_light_lc_setup_srv_t**

User data of Light LC Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting LC Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_control_t ***lc**

Parameters of the Light controller

struct **esp_ble_mesh_state_change_light_lightness_set_t**

Parameter of Light Lightness Actual state change event

Public Members

uint16_t **lightness**

The value of Light Lightness Actual state

struct **esp_ble_mesh_state_change_light_lightness_linear_set_t**

Parameter of Light Lightness Linear state change event

Public Members

uint16_t **lightness**

The value of Light Lightness Linear state

struct **esp_ble_mesh_state_change_light_lightness_default_set_t**

Parameter of Light Lightness Default state change event

Public Members

uint16_t **lightness**

The value of Light Lightness Default state

struct **esp_ble_mesh_state_change_light_lightness_range_set_t**

Parameters of Light Lightness Range state change event

Public Members

uint16_t **range_min**

The minimum value of Light Lightness Range state

uint16_t **range_max**

The maximum value of Light Lightness Range state

struct **esp_ble_mesh_state_change_light_ctl_set_t**

Parameters of Light CTL state change event

Public Members

uint16_t **lightness**

The value of Light CTL Lightness state

uint16_t **temperature**

The value of Light CTL Temperature state

int16_t **delta_uv**

The value of Light CTL Delta UV state

struct **esp_ble_mesh_state_change_light_ctl_temperature_set_t**

Parameters of Light CTL Temperature state change event

Public Members

uint16_t **temperature**

The value of Light CTL Temperature state

int16_t **delta_uv**

The value of Light CTL Delta UV state

struct **esp_ble_mesh_state_change_light_ctl_temperature_range_set_t**

Parameters of Light CTL Temperature Range state change event

Public Members

uint16_t **range_min**

The minimum value of Light CTL Temperature Range state

uint16_t **range_max**

The maximum value of Light CTL Temperature Range state

struct **esp_ble_mesh_state_change_light_ctl_default_set_t**

Parameters of Light CTL Default state change event

Public Members

uint16_t **lightness**

The value of Light Lightness Default state

uint16_t **temperature**

The value of Light CTL Temperature Default state

int16_t **delta_uv**

The value of Light CTL Delta UV Default state

struct **esp_ble_mesh_state_change_light_hsl_set_t**

Parameters of Light HSL state change event

Public Members

uint16_t **lightness**

The value of Light HSL Lightness state

uint16_t **hue**

The value of Light HSL Hue state

uint16_t **saturation**

The value of Light HSL Saturation state

struct **esp_ble_mesh_state_change_light_hsl_hue_set_t**

Parameter of Light HSL Hue state change event

Public Members

uint16_t **hue**

The value of Light HSL Hue state

struct **esp_ble_mesh_state_change_light_hsl_saturation_set_t**

Parameter of Light HSL Saturation state change event

Public Members

uint16_t **saturation**

The value of Light HSL Saturation state

struct **esp_ble_mesh_state_change_light_hsl_default_set_t**

Parameters of Light HSL Default state change event

Public Members

uint16_t **lightness**

The value of Light HSL Lightness Default state

uint16_t **hue**

The value of Light HSL Hue Default state

uint16_t **saturation**

The value of Light HSL Saturation Default state

struct **esp_ble_mesh_state_change_light_hsl_range_set_t**

Parameters of Light HSL Range state change event

Public Members

uint16_t **hue_range_min**

The minimum hue value of Light HSL Range state

uint16_t **hue_range_max**

The maximum hue value of Light HSL Range state

uint16_t **saturation_range_min**

The minimum saturation value of Light HSL Range state

uint16_t **saturation_range_max**

The maximum saturation value of Light HSL Range state

struct **esp_ble_mesh_state_change_light_xyl_set_t**

Parameters of Light xyL state change event

Public Members

uint16_t **lightness**

The value of Light xyL Lightness state

uint16_t **x**

The value of Light xyL x state

uint16_t **y**

The value of Light xyL y state

struct **esp_ble_mesh_state_change_light_xyl_default_set_t**

Parameters of Light xyL Default state change event

Public Members

uint16_t **lightness**

The value of Light Lightness Default state

uint16_t **x**

The value of Light xyL x Default state

uint16_t **y**

The value of Light xyL y Default state

struct **esp_ble_mesh_state_change_light_xyl_range_set_t**

Parameters of Light xyL Range state change event

Public Members

uint16_t **x_range_min**

The minimum value of Light xyL x Range state

uint16_t **x_range_max**

The maximum value of Light xyL x Range state

uint16_t **y_range_min**

The minimum value of Light xyL y Range state

uint16_t **y_range_max**

The maximum value of Light xyL y Range state

struct **esp_ble_mesh_state_change_light_lc_mode_set_t**

Parameter of Light LC Mode state change event

Public Members

uint8_t **mode**

The value of Light LC Mode state

struct **esp_ble_mesh_state_change_light_lc_om_set_t**

Parameter of Light LC Occupancy Mode state change event

Public Members

uint8_t **mode**

The value of Light LC Occupancy Mode state

struct **esp_ble_mesh_state_change_light_lc_light_onoff_set_t**

Parameter of Light LC Light OnOff state change event

Public Members

uint8_t **onoff**

The value of Light LC Light OnOff state

struct **esp_ble_mesh_state_change_light_lc_property_set_t**

Parameters of Light LC Property state change event

Public Members

uint16_t **property_id**

The property id of Light LC Property state

struct net_buf_simple ***property_value**

The property value of Light LC Property state

struct **esp_ble_mesh_state_change_sensor_status_t**

Parameters of Sensor Status state change event

Public Members

uint16_t **property_id**

The value of Sensor Property ID

uint8_t **occupancy**

The value of Light LC Occupancy state

uint32_t **set_occupancy_to_1_delay**

The value of Light LC Set Occupancy to 1 Delay state

uint32_t **ambient_luxlevel**

The value of Light LC Ambient Luxlevel state

union *esp_ble_mesh_state_change_sensor_status_t*::[anonymous] **state**

Parameters of Sensor Status related state

struct **esp_ble_mesh_server_recv_light_lc_property_get_t**

Context of the received Light LC Property Get message

Public Members

uint16_t **property_id**

Property ID identifying a Light LC Property

struct **esp_ble_mesh_server_recv_light_lightness_set_t**

Context of the received Light Lightness Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light lightness actual state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_lightness_linear_set_t**

Context of the received Light Lightness Linear Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light lightness linear state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_lightness_default_set_t**

Context of the received Light Lightness Default Set message

Public Members

uint16_t **lightness**

The value of the Light Lightness Default state

struct **esp_ble_mesh_server_recv_light_lightness_range_set_t**

Context of the received Light Lightness Range Set message

Public Members

uint16_t **range_min**

Value of range min field of light lightness range state

uint16_t **range_max**

Value of range max field of light lightness range state

struct **esp_ble_mesh_server_recv_light_ctl_set_t**

Context of the received Light CTL Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light ctl lightness state

uint16_t **temperature**

Target value of light ctl temperature state

int16_t **delta_uv**

Target value of light ctl delta UV state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_ctl_temperature_set_t**

Context of the received Light CTL Temperature Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **temperature**

Target value of light ctl temperature state

int16_t **delta_uv**

Target value of light ctl delta UV state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_ctl_temperature_range_set_t**

Context of the received Light CTL Temperature Range Set message

Public Members

uint16_t **range_min**

Value of temperature range min field of light ctl temperature range state

uint16_t **range_max**

Value of temperature range max field of light ctl temperature range state

struct **esp_ble_mesh_server_recv_light_ctl_default_set_t**

Context of the received Light CTL Default Set message

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **temperature**

Value of light temperature default state

int16_t **delta_uv**

Value of light delta UV default state

struct **esp_ble_mesh_server_recv_light_hsl_set_t**

Context of the received Light HSL Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **lightness**

Target value of light hsl lightness state

uint16_t **hue**

Target value of light hsl hue state

uint16_t **saturation**

Target value of light hsl saturation state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_hsl_hue_set_t**

Context of the received Light HSL Hue Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **hue**

Target value of light hsl hue state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_hsl_saturation_set_t**

Context of the received Light HSL Saturation Set message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **saturation**

Target value of light hsl hue state

uint8_t **tid**

Transaction ID

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_hsl_default_set_t**

Context of the received Light HSL Default Set message

Public Members

uint16_t **lightness**

Value of light lightness default state

uint16_t **hue**

Value of light hue default state

uint16_t **saturation**

Value of light saturation default state

struct **esp_ble_mesh_server_recv_light_hsl_range_set_t**

Context of the received Light HSL Range Set message

Public Members

uint16_t **hue_range_min**

Value of hue range min field of light hsl hue range state

uint16_t **hue_range_max**

Value of hue range max field of light hsl hue range state

uint16_t **saturation_range_min**

Value of saturation range min field of light hsl saturation range state

uint16_t **saturation_range_max**

Value of saturation range max field of light hsl saturation range state

struct **esp_ble_mesh_server_recv_light_xyl_set_t**

Context of the received Light xyL Set message

Public Members

bool **op_en**

Indicate whether optional parameters included

uint16_t **lightness**

The target value of the Light xyL Lightness state

uint16_t **x**

The target value of the Light xyL x state

uint16_t **y**

The target value of the Light xyL y state

uint8_t **tid**

Transaction Identifier

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_xyl_default_set_t**

Context of the received Light xyL Default Set message

Public Members

uint16_t **lightness**

The value of the Light Lightness Default state

uint16_t **x**

The value of the Light xyL x Default state

uint16_t **y**

The value of the Light xyL y Default state

struct **esp_ble_mesh_server_recv_light_xyl_range_set_t**

Context of the received Light xyl Range Set message

Public Members

uint16_t **x_range_min**

The value of the xyL x Range Min field of the Light xyL x Range state

uint16_t **x_range_max**

The value of the xyL x Range Max field of the Light xyL x Range state

uint16_t **y_range_min**

The value of the xyL y Range Min field of the Light xyL y Range state

uint16_t **y_range_max**

The value of the xyL y Range Max field of the Light xyL y Range state

struct **esp_ble_mesh_server_recv_light_lc_mode_set_t**

Context of the received Light LC Mode Set message

Public Members

uint8_t **mode**

The target value of the Light LC Mode state

struct **esp_ble_mesh_server_recv_light_lc_om_set_t**

Context of the received Light OM Set message

Public Members

uint8_t **mode**

The target value of the Light LC Occupancy Mode state

struct **esp_ble_mesh_server_recv_light_lc_light_onoff_set_t**

Context of the received Light LC Light OnOff Set message

Public Members

bool **op_en**

Indicate whether optional parameters included

uint8_t **light_onoff**

The target value of the Light LC Light OnOff state

uint8_t **tid**

Transaction Identifier

uint8_t **trans_time**

Time to complete state transition (optional)

uint8_t **delay**

Indicate message execution delay (C.1)

struct **esp_ble_mesh_server_recv_light_lc_property_set_t**

Context of the received Light LC Property Set message

Public Members

uint16_t **property_id**

Property ID identifying a Light LC Property

struct net_buf_simple ***property_value**

Raw value for the Light LC Property

struct **esp_ble_mesh_server_recv_sensor_status_t**

Context of the received Sensor Status message

Public Members

struct net_buf_simple ***data**

Value of sensor data state (optional)

struct **esp_ble_mesh_lighting_server_cb_param_t**

Lighting Server Model callback parameters

Public Members

esp_ble_mesh_model_t ***model**

Pointer to Lighting Server Models

esp_ble_mesh_msg_ctx_t **ctx**

Context of the received messages

esp_ble_mesh_lighting_server_cb_value_t **value**

Value of the received Lighting Messages

Macros

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_CLI(cli_pub, cli_data)

Define a new Light Lightness Client Model.

Note: This API needs to be called for each element on which the application needs to have a Light Lightness Client Model.

Parameters

- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Light Lightness Client Model instance.

ESP_BLE_MESH_MODEL_LIGHT_CTL_CLI(cli_pub, cli_data)

Define a new Light CTL Client Model.

Note: This API needs to be called for each element on which the application needs to have a Light CTL Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Light CTL Client Model instance.

ESP_BLE_MESH_MODEL_LIGHT_HSL_CLI(cli_pub, cli_data)

Define a new Light HSL Client Model.

Note: This API needs to be called for each element on which the application needs to have a Light HSL Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Light HSL Client Model instance.

ESP_BLE_MESH_MODEL_LIGHT_XYL_CLI(cli_pub, cli_data)

Define a new Light xyL Client Model.

Note: This API needs to be called for each element on which the application needs to have a Light xyL Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Light xyL Client Model instance.

ESP_BLE_MESH_MODEL_LIGHT_LC_CLI(cli_pub, cli_data)

Define a new Light LC Client Model.

Note: This API needs to be called for each element on which the application needs to have a Light LC Client Model.

Parameters
- **cli_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data** –Pointer to the unique struct *esp_ble_mesh_client_t*.

Returns New Light LC Client Model instance.

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SRV(srv_pub, srv_data)

Lighting Server Models related context.

Define a new Light Lightness Server Model.

Note: 1. The Light Lightness Server model extends the Generic Power OnOff Server model and the Generic Level Server model. When this model is present on an Element, the corresponding Light Lightness Setup Server model shall also be present.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_lightness_srv_t*.

Returns New Light Lightness Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SETUP_SRV(srv_pub, srv_data)

Define a new Light Lightness Setup Server Model.

Note: 1. The Light Lightness Setup Server model extends the Light Lightness Server model and the Generic Power OnOff Setup Server model.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_lightness_setup_srv_t*.

Returns New Light Lightness Setup Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_CTL_SRV(srv_pub, srv_data)

Define a new Light CTL Server Model.

Note: 1. The Light CTL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light CTL Temperature Server model and the corresponding Light CTL Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. The model requires two elements: the main element and the Temperature element. The Temperature element contains the corresponding Light CTL Temperature Server model and an instance of a Generic Level state bound to the Light CTL Temperature state on the Temperature element. The Light CTL Temperature state on the Temperature element is bound to the Light CTL state on the main element.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_ctl_srv_t*.

Returns New Light CTL Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_CTL_SETUP_SRV(srv_pub, srv_data)

Define a new Light CTL Setup Server Model.

Note: 1. The Light CTL Setup Server model extends the Light CTL Server and the Light Lightness Setup Server.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_ctl_setup_srv_t*.

Returns New Light CTL Setup Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_CTL_TEMP_SRV(srv_pub, srv_data)

Define a new Light CTL Temperature Server Model.

Note: 1. The Light CTL Temperature Server model extends the Generic Level Server model.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_ctl_temp_srv_t*.

Returns New Light CTL Temperature Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SRV(srv_pub, srv_data)

Define a new Light HSL Server Model.

Note: 1. The Light HSL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light HSL Hue Server model and the corresponding Light HSL Saturation Server model and the corresponding Light HSL Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. The model requires three elements: the main element and the Hue element and the Saturation element. The Hue element contains the corresponding Light HSL Hue Server model and an instance of a Generic Level state bound to the Light HSL Hue state on the Hue element. The Saturation element contains the corresponding Light HSL Saturation Server model and an instance of a Generic Level state bound to the Light HSL Saturation state on the Saturation element. The Light HSL Hue state on the Hue element is bound to the Light HSL state on the main element and the Light HSL Saturation state on the Saturation element is bound to the Light HSL state on the main element.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_hsl_srv_t*.

Returns New Light HSL Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SETUP_SRV(srv_pub, srv_data)

Define a new Light HSL Setup Server Model.

Note: 1. The Light HSL Setup Server model extends the Light HSL Server and the Light Lightness Setup Server.

a. This model shall support model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_hsl_setup_srv_t*.

Returns New Light HSL Setup Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_HSL_HUE_SRV(srv_pub, srv_data)

Define a new Light HSL Hue Server Model.

Note: 1. The Light HSL Hue Server model extends the Generic Level Server model. This model is associated with the Light HSL Server model.

a. This model shall support model publication and model subscription.

Parameters
- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_hsl_hue_srv_t*.

Returns New Light HSL Hue Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SAT_SRV(srv_pub, srv_data)

Define a new Light HSL Saturation Server Model.

Note: 1. The Light HSL Saturation Server model extends the Generic Level Server model. This model is associated with the Light HSL Server model.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_hsl_sat_srv_t*.

Returns New Light HSL Saturation Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_XYL_SRV(srv_pub, srv_data)

Define a new Light xyL Server Model.

Note: 1. The Light xyL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light xyL Setup Server model shall also be present.

a. This model shall support model publication and model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_xyl_srv_t*.

Returns New Light xyL Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_XYL_SETUP_SRV(srv_pub, srv_data)

Define a new Light xyL Setup Server Model.

Note: 1. The Light xyL Setup Server model extends the Light xyL Server and the Light Lightness Setup Server.

a. This model shall support model subscription.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_xyl_setup_srv_t*.

Returns New Light xyL Setup Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_LC_SRV(srv_pub, srv_data)

Define a new Light LC Server Model.

Note: 1. The Light LC (Lightness Control) Server model extends the Light Lightness Server model and the Generic OnOff Server model. When this model is present on an Element, the corresponding Light LC Setup Server model shall also be present.

a. This model shall support model publication and model subscription.
b. This model may be used to represent an element that is a client to a Sensor Server model and controls the Light Lightness Actual state via defined state bindings.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_lc_srv_t*.

Returns New Light LC Server Model instance.

ESP_BLE_MESH_MODEL_LIGHT_LC_SETUP_SRV(srv_pub, srv_data)

Define a new Light LC Setup Server Model.

Note: 1. The Light LC (Lightness Control) Setup model extends the Light LC Server model.

a. This model shall support model publication and model subscription.
b. This model may be used to configure setup parameters for the Light LC Server model.

Parameters

- **srv_pub** –Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **srv_data** –Pointer to the unique struct *esp_ble_mesh_light_lc_setup_srv_t*.

Returns New Light LC Setup Server Model instance.

Type Definitions

typedef void (***esp_ble_mesh_light_client_cb_t**)(*esp_ble_mesh_light_client_cb_event_t* event, *esp_ble_mesh_light_client_cb_param_t* *param)

Bluetooth Mesh Light Client Model function.

Lighting Client Model callback function type

Param event Event type
Param param Pointer to callback parameter

typedef void (***esp_ble_mesh_lighting_server_cb_t**)(*esp_ble_mesh_lighting_server_cb_event_t* event, *esp_ble_mesh_lighting_server_cb_param_t* *param)

Bluetooth Mesh Lighting Server Model function.

Lighting Server Model callback function type

Param event Event type
Param param Pointer to callback parameter

Enumerations

enum **esp_ble_mesh_light_client_cb_event_t**

This enum value is the event of Lighting Client Model

Values:

enumerator **ESP_BLE_MESH_LIGHT_CLIENT_GET_STATE_EVT**

enumerator **ESP_BLE_MESH_LIGHT_CLIENT_SET_STATE_EVT**

enumerator **ESP_BLE_MESH_LIGHT_CLIENT_PUBLISH_EVT**

enumerator **ESP_BLE_MESH_LIGHT_CLIENT_TIMEOUT_EVT**

enumerator **ESP_BLE_MESH_LIGHT_CLIENT_EVT_MAX**

enum **esp_ble_mesh_lc_state_t**

This enum value is the Light LC State Machine states

Values:

enumerator **ESP_BLE_MESH_LC_OFF**

enumerator **ESP_BLE_MESH_LC_STANDBY**

enumerator **ESP_BLE_MESH_LC_FADE_ON**

enumerator **ESP_BLE_MESH_LC_RUN**

enumerator **ESP_BLE_MESH_LC_FADE**

enumerator **ESP_BLE_MESH_LC_PROLONG**

enumerator **ESP_BLE_MESH_LC_FADE_STANDBY_AUTO**

enumerator **ESP_BLE_MESH_LC_FADE_STANDBY_MANUAL**

enum **esp_ble_mesh_lighting_server_cb_event_t**

This enum value is the event of Lighting Server Model

Values:

enumerator **ESP_BLE_MESH_LIGHTING_SERVER_STATE_CHANGE_EVT**

i. When get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, no event will be callback to the application layer when Lighting Get messages are received.
ii. When set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, this event will be callback to the application layer when Lighting Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_LIGHTING_SERVER_RECV_GET_MSG_EVT**

When get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Lighting Get messages are received.

enumerator **ESP_BLE_MESH_LIGHTING_SERVER_RECV_SET_MSG_EVT**

When set_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Lighting Set/Set Unack messages are received.

enumerator **ESP_BLE_MESH_LIGHTING_SERVER_RECV_STATUS_MSG_EVT**

When status_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Sensor Status message is received.

enumerator **ESP_BLE_MESH_LIGHTING_SERVER_EVT_MAX**

2.3.6 NimBLE-based host APIs

Overview

Apache MyNewt NimBLE is a highly configurable and BT SIG qualifiable BLE stack providing both host and controller functionalities. ESP-IDF supports NimBLE host stack which is specifically ported for ESP32 platform and FreeRTOS. The underlying controller is still the same (as in case of Bluedroid) providing VHCI interface. Refer to NimBLE user guide for a complete list of features and additional information on NimBLE stack. Most features

of NimBLE including BLE Mesh are supported by ESP-IDF. The porting layer is kept cleaner by maintaining all the existing APIs of NimBLE along with a single ESP-NimBLE API for initialization, making it simpler for the application developers.

Architecture

Currently, NimBLE host and controller support different transports such as UART and RAM between them. However, RAM transport cannot be used as is in case of ESP as ESP controller supports VHCI interface and buffering schemes used by NimBLE host is incompatible with that used by ESP controller. Therefore, a new transport between NimBLE host and ESP controller has been added. This is depicted in the figure below. This layer is responsible for maintaining pool of transport buffers and formatting buffers exchanges between host and controller as per the requirements.

Fig. 1: ESP NimBLE Stack

Threading Model

The NimBLE host can run inside the application thread or can have its own independent thread. This flexibility is inherently provided by NimBLE design. By default, a thread is spawned by the porting function `nimble_port_freertos_init`. This behavior can be changed by overriding the same function. For BLE Mesh, additional thread (advertising thread) is used which keeps on feeding advertisement events to the main thread.

Programming Sequence

To begin with, make sure that the NimBLE stack is enabled from menuconfig *choose NimBLE for the Bluetooth host*.

Typical programming sequence with NimBLE stack consists of the following steps:

- Initialize NVS flash using *`nvs_flash_init()`* API. This is because ESP controller uses NVS during initialization.
- Initialize the host and controller stack using `nimble_port_init`.
- Initialize the required NimBLE host configuration parameters and callbacks
- Perform application specific tasks/initialization
- Run the thread for host stack using `nimble_port_freertos_init`

This documentation does not cover NimBLE APIs. Refer to NimBLE tutorial for more details on the programming sequence/NimBLE APIs for different scenarios.

API Reference

Header File

- components/bt/host/nimble/esp-hci/include/esp_nimble_hci.h

Functions

esp_err_t **esp_nimble_hci_init**(void)

Initialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

This function initializes the transport buffers to be exchanged between NimBLE host and ESP controller. It also registers required host callbacks with the controller.

Returns

- ESP_OK if the initialization is successful
- Appropriate error code from esp_err_t in case of an error

esp_err_t **esp_nimble_hci_deinit**(void)

Deinitialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

Note: This function should be called after the NimBLE host is deinitialized.

Returns

- ESP_OK if the deinitialization is successful
- Appropriate error codes from esp_err_t in case of an error

Macros

BLE_HCI_UART_H4_NONE

BLE_HCI_UART_H4_CMD

BLE_HCI_UART_H4_ACL

BLE_HCI_UART_H4_SCO

BLE_HCI_UART_H4_EVT

ESP-IDF currently supports two host stacks. The Bluedroid based stack (default) supports classic Bluetooth as well as BLE. On the other hand, Apache NimBLE based stack is BLE only. For users to make a choice:

- For usecases involving classic Bluetooth as well as BLE, Bluedroid should be used.
- For BLE-only usecases, using NimBLE is recommended. It is less demanding in terms of code footprint and runtime memory, making it suitable for such scenarios.

For the overview of the ESP32 Bluetooth stack architecture, follow the links below:

- ESP32 Bluetooth Architecture (PDF)

Code examples for this API section are provided in the bluetooth/bluedroid directory of ESP-IDF examples.

The following examples contain detailed walkthroughs:

- GATT Client Example Walkthrough
- GATT Server Service Table Example Walkthrough
- GATT Server Example Walkthrough
- GATT Security Client Example Walkthrough
- GATT Security Server Example Walkthrough
- GATT Client Multi-connection Example Walkthrough

2.4 Error Codes Reference

This section lists various error code constants defined in ESP-IDF.

For general information about error codes in ESP-IDF, see *Error Handling*.

ESP_FAIL (-1): Generic esp_err_t code indicating failure

ESP_OK (0): esp_err_t value indicating success (no error)

ESP_ERR_NO_MEM **(0x101)**: Out of memory

ESP_ERR_INVALID_ARG **(0x102)**: Invalid argument

ESP_ERR_INVALID_STATE **(0x103)**: Invalid state

ESP_ERR_INVALID_SIZE **(0x104)**: Invalid size

ESP_ERR_NOT_FOUND **(0x105)**: Requested resource not found

ESP_ERR_NOT_SUPPORTED **(0x106)**: Operation or feature not supported

ESP_ERR_TIMEOUT **(0x107)**: Operation timed out

ESP_ERR_INVALID_RESPONSE **(0x108)**: Received response was invalid

ESP_ERR_INVALID_CRC **(0x109)**: CRC or checksum was invalid

ESP_ERR_INVALID_VERSION **(0x10a)**: Version was invalid

ESP_ERR_INVALID_MAC **(0x10b)**: MAC address was invalid

ESP_ERR_NOT_FINISHED **(0x10c)**: There are items remained to retrieve

ESP_ERR_NVS_BASE **(0x1100)**: Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED **(0x1101)**: The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND **(0x1102)**: A requested entry couldn't be found or namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH **(0x1103)**: The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY **(0x1104)**: Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE **(0x1105)**: There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME **(0x1106)**: Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE **(0x1107)**: Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED **(0x1108)**: The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG **(0x1109)**: Key name is too long

ESP_ERR_NVS_PAGE_FULL **(0x110a)**: Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE **(0x110b)**: NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH **(0x110c)**: String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES **(0x110d)**: NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG **(0x110e)**: Value doesn't fit into the entry or string or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND **(0x110f)**: Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND **(0x1110)**: NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED **(0x1111)**: XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED **(0x1112)**: XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED **(0x1113)**: XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND **(0x1114)**: XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED **(0x1115)**: NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED **(0x1116)**: NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART **(0x1117)**: NVS key partition is corrupt

ESP_ERR_NVS_CONTENT_DIFFERS **(0x1118)**: Internal error; never returned by nvs API functions. NVS key is different in comparison

ESP_ERR_NVS_WRONG_ENCRYPTION **(0x1119)**: NVS partition is marked as encrypted with generic flash encryption. This is forbidden since the NVS encryption works differently.

ESP_ERR_ULP_BASE **(0x1200)**: Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG **(0x1201)**: Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR **(0x1202)**: Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL **(0x1203)**: More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL **(0x1204)**: Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE **(0x1205)**: Branch target is out of range of B instruction (try replacing with BX)

ESP_ERR_OTA_BASE **(0x1500)**: Base error code for ota_ops api

ESP_ERR_OTA_PARTITION_CONFLICT **(0x1501)**: Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID **(0x1502)**: Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED **(0x1503)**: Error if OTA app image is invalid

ESP_ERR_OTA_SMALL_SEC_VER **(0x1504)**: Error if the firmware has a secure version less than the running firmware.

ESP_ERR_OTA_ROLLBACK_FAILED **(0x1505)**: Error if flash does not have valid firmware in passive partition and hence rollback is not possible

ESP_ERR_OTA_ROLLBACK_INVALID_STATE **(0x1506)**: Error if current active firmware is still marked in pending validation state (ESP_OTA_IMG_PENDING_VERIFY), essentially first boot of firmware image post upgrade and hence firmware upgrade is not possible

ESP_ERR_EFUSE **(0x1600)**: Base error code for efuse api.

ESP_OK_EFUSE_CNT **(0x1601)**: OK the required number of bits is set.

ESP_ERR_EFUSE_CNT_IS_FULL **(0x1602)**: Error field is full.

ESP_ERR_EFUSE_REPEATED_PROG **(0x1603)**: Error repeated programming of programmed bits is strictly forbidden.

ESP_ERR_CODING **(0x1604)**: Error while a encoding operation.

ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS **(0x1605)**: Error not enough unused key blocks available

ESP_ERR_DAMAGED_READING **(0x1606)**: Error. Burn or reset was done during a reading operation leads to damage read data. This error is internal to the efuse component and not returned by any public API.

ESP_ERR_IMAGE_BASE **(0x2000)**

ESP_ERR_IMAGE_FLASH_FAIL **(0x2001)**

ESP_ERR_IMAGE_INVALID **(0x2002)**

ESP_ERR_WIFI_BASE **(0x3000)**: Starting number of WiFi error codes

ESP_ERR_WIFI_NOT_INIT **(0x3001)**: WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED **(0x3002)**: WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED **(0x3003)**: WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF **(0x3004)**: WiFi interface error

ESP_ERR_WIFI_MODE **(0x3005)**: WiFi mode error

ESP_ERR_WIFI_STATE **(0x3006)**: WiFi internal state error

ESP_ERR_WIFI_CONN **(0x3007)**: WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS **(0x3008)**: WiFi internal NVS module error

ESP_ERR_WIFI_MAC **(0x3009)**: MAC address is invalid

ESP_ERR_WIFI_SSID **(0x300a)**: SSID is invalid

ESP_ERR_WIFI_PASSWORD **(0x300b)**: Password is invalid

ESP_ERR_WIFI_TIMEOUT **(0x300c)**: Timeout error

ESP_ERR_WIFI_WAKE_FAIL **(0x300d)**: WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK **(0x300e)**: The caller would block

ESP_ERR_WIFI_NOT_CONNECT **(0x300f)**: Station still in disconnect status

ESP_ERR_WIFI_POST **(0x3012)**: Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE **(0x3013)**: Invalid WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE **(0x3014)**: Returned when WiFi is stopping

ESP_ERR_WIFI_NOT_ASSOC **(0x3015)**: The WiFi connection is not associated

ESP_ERR_WIFI_TX_DISALLOW **(0x3016)**: The WiFi TX is disallowed

ESP_ERR_WIFI_DISCARD **(0x3017)**: Discard frame

ESP_ERR_WIFI_ROC_IN_PROGRESS **(0x3018)**: ROC op is in progress

ESP_ERR_WIFI_REGISTRAR **(0x3033)**: WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE **(0x3034)**: WPS type error

ESP_ERR_WIFI_WPS_SM **(0x3035)**: WPS state machine is not initialized

ESP_ERR_ESPNOW_BASE **(0x3064)**: ESPNOW error number base.

ESP_ERR_ESPNOW_NOT_INIT **(0x3065)**: ESPNOW is not initialized.

ESP_ERR_ESPNOW_ARG **(0x3066)**: Invalid argument

ESP_ERR_ESPNOW_NO_MEM **(0x3067)**: Out of memory

ESP_ERR_ESPNOW_FULL **(0x3068)**: ESPNOW peer list is full

ESP_ERR_ESPNOW_NOT_FOUND **(0x3069)**: ESPNOW peer is not found

ESP_ERR_ESPNOW_INTERNAL **(0x306a)**: Internal error

ESP_ERR_ESPNOW_EXIST **(0x306b)**: ESPNOW peer has existed

ESP_ERR_ESPNOW_IF **(0x306c)**: Interface error

ESP_ERR_DPP_FAILURE **(0x3097)**: Generic failure during DPP Operation

ESP_ERR_DPP_TX_FAILURE **(0x3098)**: DPP Frame Tx failed OR not Acked

ESP_ERR_DPP_INVALID_ATTR **(0x3099)**: Encountered invalid DPP Attribute

ESP_ERR_DPP_AUTH_TIMEOUT **(0x309a)**: DPP Auth response was not recieved in time

ESP_ERR_MESH_BASE **(0x4000)**: Starting number of MESH error codes

ESP_ERR_MESH_WIFI_NOT_START **(0x4001)**

ESP_ERR_MESH_NOT_INIT **(0x4002)**

ESP_ERR_MESH_NOT_CONFIG **(0x4003)**

ESP_ERR_MESH_NOT_START **(0x4004)**

ESP_ERR_MESH_NOT_SUPPORT **(0x4005)**

ESP_ERR_MESH_NOT_ALLOWED **(0x4006)**

ESP_ERR_MESH_NO_MEMORY **(0x4007)**

ESP_ERR_MESH_ARGUMENT **(0x4008)**

ESP_ERR_MESH_EXCEED_MTU **(0x4009)**

ESP_ERR_MESH_TIMEOUT **(0x400a)**

ESP_ERR_MESH_DISCONNECTED **(0x400b)**

ESP_ERR_MESH_QUEUE_FAIL **(0x400c)**

ESP_ERR_MESH_QUEUE_FULL **(0x400d)**

ESP_ERR_MESH_NO_PARENT_FOUND **(0x400e)**

ESP_ERR_MESH_NO_ROUTE_FOUND **(0x400f)**

ESP_ERR_MESH_OPTION_NULL **(0x4010)**

ESP_ERR_MESH_OPTION_UNKNOWN **(0x4011)**

ESP_ERR_MESH_XON_NO_WINDOW **(0x4012)**

ESP_ERR_MESH_INTERFACE **(0x4013)**

ESP_ERR_MESH_DISCARD_DUPLICATE **(0x4014)**

ESP_ERR_MESH_DISCARD **(0x4015)**

ESP_ERR_MESH_VOTING **(0x4016)**

ESP_ERR_MESH_XMIT **(0x4017)**

ESP_ERR_MESH_QUEUE_READ **(0x4018)**

ESP_ERR_MESH_PS **(0x4019)**

ESP_ERR_MESH_RECV_RELEASE **(0x401a)**

ESP_ERR_ESP_NETIF_BASE **(0x5000)**

ESP_ERR_ESP_NETIF_INVALID_PARAMS **(0x5001)**

ESP_ERR_ESP_NETIF_IF_NOT_READY **(0x5002)**

ESP_ERR_ESP_NETIF_DHCPC_START_FAILED **(0x5003)**

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED **(0x5004)**

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED **(0x5005)**

ESP_ERR_ESP_NETIF_NO_MEM **(0x5006)**

ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED **(0x5007)**

ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED **(0x5008)**

ESP_ERR_ESP_NETIF_INIT_FAILED **(0x5009)**

ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED **(0x500a)**

ESP_ERR_ESP_NETIF_MLD6_FAILED **(0x500b)**

ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED **(0x500c)**

ESP_ERR_ESP_NETIF_DHCPS_START_FAILED **(0x500d)**

ESP_ERR_FLASH_BASE **(0x6000)**: Starting number of flash error codes

ESP_ERR_FLASH_OP_FAIL **(0x6001)**

ESP_ERR_FLASH_OP_TIMEOUT **(0x6002)**

ESP_ERR_FLASH_NOT_INITIALISED **(0x6003)**

ESP_ERR_FLASH_UNSUPPORTED_HOST **(0x6004)**

ESP_ERR_FLASH_UNSUPPORTED_CHIP **(0x6005)**

ESP_ERR_FLASH_PROTECTED **(0x6006)**

ESP_ERR_HTTP_BASE **(0x7000)**: Starting number of HTTP error codes

ESP_ERR_HTTP_MAX_REDIRECT **(0x7001)**: The error exceeds the number of HTTP redirects

ESP_ERR_HTTP_CONNECT **(0x7002)**: Error open the HTTP connection

ESP_ERR_HTTP_WRITE_DATA **(0x7003)**: Error write HTTP data

ESP_ERR_HTTP_FETCH_HEADER **(0x7004)**: Error read HTTP header from server

ESP_ERR_HTTP_INVALID_TRANSPORT **(0x7005)**: There are no transport support for the input scheme

ESP_ERR_HTTP_CONNECTING **(0x7006)**: HTTP connection hasn't been established yet

ESP_ERR_HTTP_EAGAIN **(0x7007)**: Mapping of errno EAGAIN to esp_err_t

ESP_ERR_HTTP_CONNECTION_CLOSED **(0x7008)**: Read FIN from peer and the connection closed

ESP_ERR_ESP_TLS_BASE **(0x8000)**: Starting number of ESP-TLS error codes

ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME **(0x8001)**: Error if hostname couldn't be resolved upon tls connection

ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET **(0x8002)**: Failed to create socket

ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY **(0x8003)**: Unsupported protocol family

ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST **(0x8004)**: Failed to connect to host

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED **(0x8005)**: failed to set/get socket option

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT **(0x8006)**: new connection in esp_tls_low_level_conn connection timeouted

ESP_ERR_ESP_TLS_SE_FAILED **(0x8007)**

ESP_ERR_ESP_TLS_TCP_CLOSED_FIN **(0x8008)**

ESP_ERR_MBEDTLS_CERT_PARTLY_OK **(0x8010)**: mbedtls parse certificates was partly successful

ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED **(0x8011)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED **(0x8012)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED **(0x8013)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED **(0x8014)**: mbedtls api returned error

ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED **(0x8015)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED **(0x8016)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SETUP_FAILED **(0x8017)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_WRITE_FAILED **(0x8018)**: mbedtls api returned error

ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED **(0x8019)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED **(0x801a)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED **(0x801b)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED **(0x801c)**: mbedtls api returned failed

ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED **(0x8031)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED **(0x8032)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED **(0x8033)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED **(0x8034)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED **(0x8035)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_CTX_SETUP_FAILED **(0x8036)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_SETUP_FAILED **(0x8037)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_WRITE_FAILED **(0x8038)**: wolfSSL api returned failed

ESP_ERR_HTTPS_OTA_BASE **(0x9000)**

ESP_ERR_HTTPS_OTA_IN_PROGRESS **(0x9001)**

ESP_ERR_PING_BASE **(0xa000)**

ESP_ERR_PING_INVALID_PARAMS **(0xa001)**

ESP_ERR_PING_NO_MEM **(0xa002)**

ESP_ERR_HTTPD_BASE **(0xb000)**: Starting number of HTTPD error codes

ESP_ERR_HTTPD_HANDLERS_FULL **(0xb001)**: All slots for registering URI handlers have been consumed

ESP_ERR_HTTPD_HANDLER_EXISTS **(0xb002)**: URI handler with same method and target URI already registered

ESP_ERR_HTTPD_INVALID_REQ **(0xb003)**: Invalid request pointer

ESP_ERR_HTTPD_RESULT_TRUNC **(0xb004)**: Result string truncated

ESP_ERR_HTTPD_RESP_HDR **(0xb005)**: Response header field larger than supported

ESP_ERR_HTTPD_RESP_SEND **(0xb006)**: Error occured while sending response packet

ESP_ERR_HTTPD_ALLOC_MEM **(0xb007)**: Failed to dynamically allocate memory for resource

ESP_ERR_HTTPD_TASK **(0xb008)**: Failed to launch server task/thread

ESP_ERR_HW_CRYPTO_BASE **(0xc000)**: Starting number of HW cryptography module error codes

ESP_ERR_HW_CRYPTO_DS_HMAC_FAIL **(0xc001)**: HMAC peripheral problem

ESP_ERR_HW_CRYPTO_DS_INVALID_KEY **(0xc002)**

ESP_ERR_HW_CRYPTO_DS_INVALID_DIGEST **(0xc004)**

ESP_ERR_HW_CRYPTO_DS_INVALID_PADDING **(0xc005)**

ESP_ERR_MEMPROT_BASE **(0xd000)**: Starting number of Memory Protection API error codes

ESP_ERR_MEMPROT_MEMORY_TYPE_INVALID **(0xd001)**

ESP_ERR_MEMPROT_SPLIT_ADDR_INVALID **(0xd002)**

ESP_ERR_MEMPROT_SPLIT_ADDR_OUT_OF_RANGE **(0xd003)**

ESP_ERR_MEMPROT_SPLIT_ADDR_UNALIGNED **(0xd004)**

ESP_ERR_MEMPROT_UNIMGMT_BLOCK_INVALID **(0xd005)**

ESP_ERR_MEMPROT_WORLD_INVALID **(0xd006)**

ESP_ERR_MEMPROT_AREA_INVALID **(0xd007)**

ESP_ERR_MEMPROT_CPUID_INVALID **(0xd008)**

ESP_ERR_TCP_TRANSPORT_BASE **(0xe000)**: Starting number of TCP Transport error codes

ESP_ERR_TCP_TRANSPORT_CONNECTION_TIMEOUT **(0xe001)**: Connection has timed out

`ESP_ERR_TCP_TRANSPORT_CONNECTION_CLOSED_BY_FIN` **(0xe002)**: Read FIN from peer and the connection has closed (in a clean way)

`ESP_ERR_TCP_TRANSPORT_CONNECTION_FAILED` **(0xe003)**: Failed to connect to the peer

`ESP_ERR_TCP_TRANSPORT_NO_MEM` **(0xe004)**: Memory allocation failed

2.5 Networking APIs

2.5.1 Wi-Fi

ESP-NOW

Overview ESP-NOW is a kind of connectionless Wi-Fi communication protocol that is defined by Espressif. In ESP-NOW, application data is encapsulated in a vendor-specific action frame and then transmitted from one Wi-Fi device to another without connection. CTR with CBC-MAC Protocol(CCMP) is used to protect the action frame for security. ESP-NOW is widely used in smart light, remote controlling, sensor, etc.

Frame Format ESP-NOW uses a vendor-specific action frame to transmit ESP-NOW data. The default ESP-NOW bit rate is 1 Mbps. The format of the vendor-specific action frame is as follows:

```
------------------------------------------------------------------------------------------
↪-------------------------
| MAC Header | Category Code | Organization Identifier | Random Values | Vendor␣
↪Specific Content |   FCS   |
------------------------------------------------------------------------------------------
↪-------------------------
  24 bytes         1 byte              3 bytes               4 bytes             7~
↪257 bytes        4 bytes
```

- Category Code: The Category Code field is set to the value(127) indicating the vendor-specific category.
- Organization Identifier: The Organization Identifier contains a unique identifier (0x18fe34), which is the first three bytes of MAC address applied by Espressif.
- Random Value: The Random Value filed is used to prevents relay attacks.
- Vendor Specific Content: The Vendor Specific Content contains vendor-specific fields as follows:

```
-------------------------------------------------------------------------------
| Element ID | Length | Organization Identifier | Type | Version |    Body    |
-------------------------------------------------------------------------------
    1 byte     1 byte            3 bytes         1 byte   1 byte   0~250 bytes
```

- Element ID: The Element ID field is set to the value (221), indicating the vendor-specific element.
- Length: The length is the total length of Organization Identifier, Type, Version and Body.
- Organization Identifier: The Organization Identifier contains a unique identifier(0x18fe34), which is the first three bytes of MAC address applied by Espressif.
- Type: The Type field is set to the value (4) indicating ESP-NOW.
- Version: The Version field is set to the version of ESP-NOW.
- Body: The Body contains the ESP-NOW data.

As ESP-NOW is connectionless, the MAC header is a little different from that of standard frames. The FromDS and ToDS bits of FrameControl field are both 0. The first address field is set to the destination address. The second address field is set to the source address. The third address field is set to broadcast address (0xff:0xff:0xff:0xff:0xff:0xff).

Security

ESP-NOW uses the CCMP method, which is described in IEEE Std. 802.11-2012, to protect the vendor-specific action frame

- PMK is used to encrypt LMK with the AES-128 algorithm. Call `esp_now_set_pmk()` to set PMK. If PMK is not set, a default PMK will be used.
- LMK of the paired device is used to encrypt the vendor-specific action frame with the CCMP method. If the LMK of the paired device is not set, the vendor-specific action frame will not be encrypted.

Encrypting multicast vendor-specific action frame is not supported.

Initialization and De-initialization Call `esp_now_init()` to initialize ESP-NOW and `esp_now_deinit()` to de-initialize ESP-NOW. ESP-NOW data must be transmitted after Wi-Fi is started, so it is recommended to start Wi-Fi before initializing ESP-NOW and stop Wi-Fi after de-initializing ESP-NOW. When `esp_now_deinit()` is called, all of the information of paired devices will be deleted.

Add Paired Device Call `esp_now_add_peer()` to add the device to the paired device list before you send data to this device. If security is enabled, the LMK must be set. You can send ESP-NOW data via both the Station and the SoftAP interface. Make sure that the interface is enabled before sending ESP-NOW data.

The maximum number of paired devices is 20, and the paired encryption devices are no more than 17, the default is 7. If you want to change the number of paired encryption devices, set *CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM* in the Wi-Fi component configuration menu.

A device with a broadcast MAC address must be added before sending broadcast data. The range of the channel of paired devices is from 0 to 14. If the channel is set to 0, data will be sent on the current channel. Otherwise, the channel must be set as the channel that the local device is on.

Send ESP-NOW Data Call `esp_now_send()` to send ESP-NOW data and `esp_now_register_send_cb()` to register sending callback function. It will return *ESP_NOW_SEND_SUCCESS* in sending callback function if the data is received successfully on the MAC layer. Otherwise, it will return *ESP_NOW_SEND_FAIL*. Several reasons can lead to ESP-NOW fails to send data. For example, the destination device doesn't exist; the channels of the devices are not the same; the action frame is lost when transmitting on the air, etc. It is not guaranteed that application layer can receive the data. If necessary, send back ack data when receiving ESP-NOW data. If receiving ack data timeouts, retransmit the ESP-NOW data. A sequence number can also be assigned to ESP-NOW data to drop the duplicate data.

If there is a lot of ESP-NOW data to send, call `esp_now_send()` to send less than or equal to 250 bytes of data once a time. Note that too short interval between sending two ESP-NOW data may lead to disorder of sending callback function. So, it is recommended that sending the next ESP-NOW data after the sending callback function of the previous sending has returned. The sending callback function runs from a high-priority Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post the necessary data to a queue and handle it from a lower priority task.

Receiving ESP-NOW Data Call `esp_now_register_recv_cb()` to register receiving callback function. Call the receiving callback function when receiving ESP-NOW. The receiving callback function also runs from the Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post the necessary data to a queue and handle it from a lower priority task.

Config ESP-NOW Rate Call `esp_wifi_config_espnow_rate()` to config ESPNOW rate of specified interface. Make sure that the interface is enabled before config rate. This API should be called after `esp_wifi_start()`.

Config ESP-NOW Power-saving Parameter Sleep is supported only when ESP32 is configured as station.

Call `esp_now_set_wake_window()` to configure Window for ESP-NOW RX at sleep. The default value is the maximum, which allowing RX all the time.

If Power-saving is needed for ESP-NOW, call *esp_wifi_connectionless_module_set_wake_interval()* to configure Interval as well.

Please refer to *connectionless module power save* to get more detail.

Application Examples

- Example of sending and receiving ESP-NOW data between two devices: wifi/espnow.
- For more application examples of how to use ESP-NOW, please visit ESP-NOW repository.

API Reference

Header File

- components/esp_wifi/include/esp_now.h

Functions

esp_err_t **esp_now_init** (void)

Initialize ESPNOW function.

Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_INTERNAL : Internal error

esp_err_t **esp_now_deinit** (void)

De-initialize ESPNOW function.

Returns
- ESP_OK : succeed

esp_err_t **esp_now_get_version** (uint32_t *version)

Get the version of ESPNOW.

Parameters **version** –ESPNOW version
Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_ARG : invalid argument

esp_err_t **esp_now_register_recv_cb** (*esp_now_recv_cb_t* cb)

Register callback function of receiving ESPNOW data.

Parameters **cb** –callback function of receiving ESPNOW data
Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_INTERNAL : internal error

esp_err_t **esp_now_unregister_recv_cb** (void)

Unregister callback function of receiving ESPNOW data.

Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

esp_err_t **esp_now_register_send_cb** (*esp_now_send_cb_t* cb)

Register callback function of sending ESPNOW data.

Parameters **cb** –callback function of sending ESPNOW data
Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

- ESP_ERR_ESPNOW_INTERNAL : internal error

esp_err_t **esp_now_unregister_send_cb**(void)

Unregister callback function of sending ESPNOW data.

Returns

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

esp_err_t **esp_now_send**(const uint8_t *peer_addr, const uint8_t *data, size_t len)

Send ESPNOW data.

Attention 1. If peer_addr is not NULL, send data to the peer whose MAC address matches peer_addr
Attention 2. If peer_addr is NULL, send data to all of the peers that are added to the peer list
Attention 3. The maximum length of data must be less than ESP_NOW_MAX_DATA_LEN
Attention 4. The buffer pointed to by data argument does not need to be valid after esp_now_send returns

Parameters

- **peer_addr** –peer MAC address
- **data** –data to send
- **len** –length of data

Returns

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_INTERNAL : internal error
- ESP_ERR_ESPNOW_NO_MEM : out of memory, when this happens, you can delay a while before sending the next data
- ESP_ERR_ESPNOW_NOT_FOUND : peer is not found
- ESP_ERR_ESPNOW_IF : current WiFi interface doesn't match that of peer

esp_err_t **esp_now_add_peer**(const *esp_now_peer_info_t* *peer)

Add a peer to peer list.

Parameters **peer** –peer information

Returns

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_FULL : peer list is full
- ESP_ERR_ESPNOW_NO_MEM : out of memory
- ESP_ERR_ESPNOW_EXIST : peer has existed

esp_err_t **esp_now_del_peer**(const uint8_t *peer_addr)

Delete a peer from peer list.

Parameters **peer_addr** –peer MAC address

Returns

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_NOT_FOUND : peer is not found

esp_err_t **esp_now_mod_peer**(const *esp_now_peer_info_t* *peer)

Modify a peer.

Parameters **peer** –peer information

Returns

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_FULL : peer list is full

esp_err_t **esp_wifi_config_espnow_rate**(*wifi_interface_t* ifx, *wifi_phy_rate_t* rate)

Config ESPNOW rate of specified interface.

Attention 1. This API should be called after esp_wifi_start().

Parameters
- **ifx** –Interface to be configured.
- **rate** –Phy rate to be configured.

Returns
- ESP_OK: succeed
- others: failed

esp_err_t **esp_now_get_peer**(const uint8_t *peer_addr, *esp_now_peer_info_t* *peer)

Get a peer whose MAC address matches peer_addr from peer list.

Parameters
- **peer_addr** –peer MAC address
- **peer** –peer information

Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_NOT_FOUND : peer is not found

esp_err_t **esp_now_fetch_peer**(bool from_head, *esp_now_peer_info_t* *peer)

Fetch a peer from peer list. Only return the peer which address is unicast, for the multicast/broadcast address, the function will ignore and try to find the next in the peer list.

Parameters
- **from_head** –fetch from head of list or not
- **peer** –peer information

Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_NOT_FOUND : peer is not found

bool **esp_now_is_peer_exist**(const uint8_t *peer_addr)

Peer exists or not.

Parameters **peer_addr** –peer MAC address

Returns
- true : peer exists
- false : peer not exists

esp_err_t **esp_now_get_peer_num**(*esp_now_peer_num_t* *num)

Get the number of peers.

Parameters **num** –number of peers

Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument

esp_err_t **esp_now_set_pmk**(const uint8_t *pmk)

Set the primary master key.

Attention 1. primary master key is used to encrypt local master key

Parameters **pmk** –primary master key
Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument

esp_err_t **esp_now_set_wake_window**(uint16_t window)

Set wake window for esp_now to wake up in interval unit.

Attention 1. This configuration could work at connected status. When ESP_WIFI_STA_DISCONNECTED_PM_ENABLE is enabled, this configuration could work at disconnected status.
Attention 2. Default value is the maximum.

Parameters **window** –Milliseconds would the chip keep waked each interval, from 0 to 65535.
Returns
- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

Structures

struct **esp_now_peer_info**

ESPNOW peer information parameters.

Public Members

uint8_t **peer_addr**[ESP_NOW_ETH_ALEN]

ESPNOW peer MAC address that is also the MAC address of station or softap

uint8_t **lmk**[ESP_NOW_KEY_LEN]

ESPNOW peer local master key that is used to encrypt data

uint8_t **channel**

Wi-Fi channel that peer uses to send/receive ESPNOW data. If the value is 0, use the current channel which station or softap is on. Otherwise, it must be set as the channel that station or softap is on.

wifi_interface_t **ifidx**

Wi-Fi interface that peer uses to send/receive ESPNOW data

bool **encrypt**

ESPNOW data that this peer sends/receives is encrypted or not

void ***priv**

ESPNOW peer private data

struct **esp_now_peer_num**

Number of ESPNOW peers which exist currently.

Public Members

int **total_num**

Total number of ESPNOW peers, maximum value is ESP_NOW_MAX_TOTAL_PEER_NUM

int **encrypt_num**

Number of encrypted ESPNOW peers, maximum value is ESP_NOW_MAX_ENCRYPT_PEER_NUM

struct **esp_now_recv_info**

ESPNOW packet information.

Public Members

uint8_t ***src_addr**

Source address of ESPNOW packet

uint8_t ***des_addr**

Destination address of ESPNOW packet

wifi_pkt_rx_ctrl_t ***rx_ctrl**

Rx control info of ESPNOW packet

Macros

ESP_ERR_ESPNOW_BASE

ESPNOW error number base.

ESP_ERR_ESPNOW_NOT_INIT

ESPNOW is not initialized.

ESP_ERR_ESPNOW_ARG

Invalid argument

ESP_ERR_ESPNOW_NO_MEM

Out of memory

ESP_ERR_ESPNOW_FULL

ESPNOW peer list is full

ESP_ERR_ESPNOW_NOT_FOUND

ESPNOW peer is not found

ESP_ERR_ESPNOW_INTERNAL

Internal error

ESP_ERR_ESPNOW_EXIST

ESPNOW peer has existed

ESP_ERR_ESPNOW_IF

Interface error

ESP_NOW_ETH_ALEN

Length of ESPNOW peer MAC address

ESP_NOW_KEY_LEN

Length of ESPNOW peer local master key

ESP_NOW_MAX_TOTAL_PEER_NUM

Maximum number of ESPNOW total peers

ESP_NOW_MAX_ENCRYPT_PEER_NUM

Maximum number of ESPNOW encrypted peers

ESP_NOW_MAX_DATA_LEN

Maximum length of ESPNOW data which is sent very time

Type Definitions

typedef struct *esp_now_peer_info* **esp_now_peer_info_t**

ESPNOW peer information parameters.

typedef struct *esp_now_peer_num* **esp_now_peer_num_t**

Number of ESPNOW peers which exist currently.

typedef struct *esp_now_recv_info* **esp_now_recv_info_t**

ESPNOW packet information.

typedef void (***esp_now_recv_cb_t**)(const *esp_now_recv_info_t* *esp_now_info, const uint8_t *data, int data_len)

Callback function of receiving ESPNOW data.

Attention esp_now_info is a local variable, it can only be used in the callback.

Param esp_now_info received ESPNOW packet information
Param data received data
Param data_len length of received data

typedef void (***esp_now_send_cb_t**)(const uint8_t *mac_addr, *esp_now_send_status_t* status)

Callback function of sending ESPNOW data.

Param mac_addr peer MAC address
Param status status of sending ESPNOW data (succeed or fail)

Enumerations

enum **esp_now_send_status_t**

Status of sending ESPNOW data .

Values:

enumerator **ESP_NOW_SEND_SUCCESS**

Send ESPNOW data successfully

enumerator **ESP_NOW_SEND_FAIL**

Send ESPNOW data fail

ESP-WIFI-MESH Programming Guide

This is a programming guide for ESP-WIFI-MESH, including the API reference and coding examples. This guide is split into the following parts:

1. *ESP-WIFI-MESH Programming Model*
2. *Writing an ESP-WIFI-MESH Application*
3. *Self Organized Networking*
4. *Application Examples*
5. *API Reference*

For documentation regarding the ESP-WIFI-MESH protocol, please see the *ESP-WIFI-MESH API Guide*. For more information about ESP-WIFI-MESH Development Framework, please see ESP-WIFI-MESH Development Framework.

ESP-WIFI-MESH Programming Model

Software Stack The ESP-WIFI-MESH software stack is built atop the Wi-Fi Driver/FreeRTOS and may use the LwIP Stack in some instances (i.e. the root node). The following diagram illustrates the ESP-WIFI-MESH software stack.

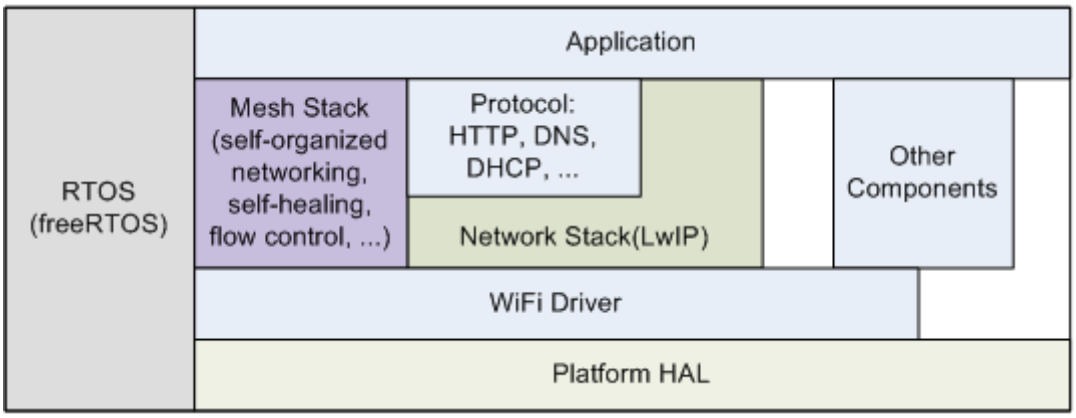


Fig. 2: ESP-WIFI-MESH Software Stack

System Events An application interfaces with ESP-WIFI-MESH via **ESP-WIFI-MESH Events**. Since ESP-WIFI-MESH is built atop the Wi-Fi stack, it is also possible for the application to interface with the Wi-Fi driver via the **Wi-Fi Event Task**. The following diagram illustrates the interfaces for the various System Events in an ESP-WIFI-MESH application.

The `mesh_event_id_t` defines all possible ESP-WIFI-MESH events and can indicate events such as the connection/disconnection of parent/child. Before ESP-WIFI-MESH events can be used, the application must register a **Mesh Events handler** via `esp_event_handler_register()` to the default event task. The Mesh Events handler that is registered contain handlers for each ESP-WIFI-MESH event relevant to the application.

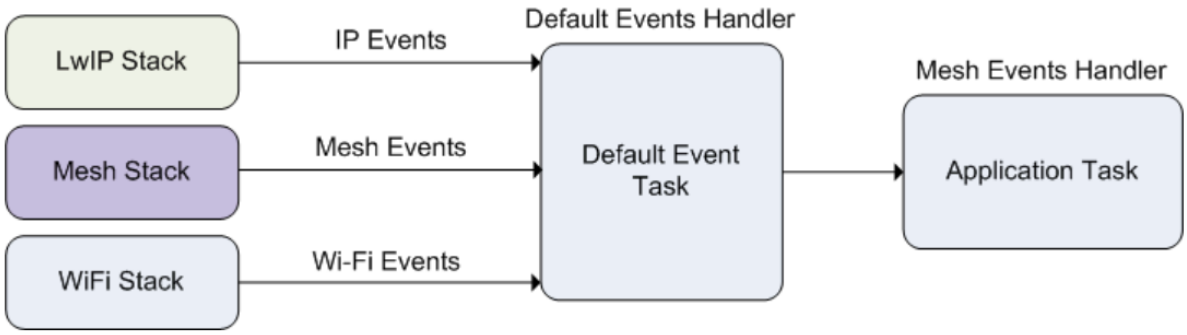


Fig. 3: ESP-WIFI-MESH System Events Delivery

Typical use cases of mesh events include using events such as *MESH_EVENT_PARENT_CONNECTED* and *MESH_EVENT_CHILD_CONNECTED* to indicate when a node can begin transmitting data upstream and downstream respectively. Likewise, *IP_EVENT_STA_GOT_IP* and *IP_EVENT_STA_LOST_IP* can be used to indicate when the root node can and cannot transmit data to the external IP network.

Warning: When using ESP-WIFI-MESH under self-organized mode, users must ensure that no calls to Wi-Fi API are made. This is due to the fact that the self-organizing mode will internally make Wi-Fi API calls to connect/disconnect/scan etc. **Any Wi-Fi calls from the application (including calls from callbacks and handlers of Wi-Fi events) may interfere with ESP-WIFI-MESH' s self-organizing behavior**. Therefore, users should not call Wi-Fi APIs after *esp_mesh_start()* is called, and before *esp_mesh_stop()* is called.

LwIP & ESP-WIFI-MESH The application can access the ESP-WIFI-MESH stack directly without having to go through the LwIP stack. The LwIP stack is only required by the root node to transmit/receive data to/from an external IP network. However, since every node can potentially become the root node (due to automatic root node selection), each node must still initialize the LwIP stack.

Each node that could become root is required to initialize LwIP by calling *esp_netif_init()*. In order to prevent non-root node access to LwIP, the application should not create or register any network interfaces using esp_netif APIs.

> ESP-WIFI-MESH requires a root node to be connected with a router. Therefore, in the event that a node becomes the root, **the corresponding handler must start the DHCP client service and immediately obtain an IP address**. Doing so will allow other nodes to begin transmitting/receiving packets to/from the external IP network. However, this step is unnecessary if static IP settings are used.

Writing an ESP-WIFI-MESH Application The prerequisites for starting ESP-WIFI-MESH is to initialize LwIP and Wi-Fi, The following code snippet demonstrates the necessary prerequisite steps before ESP-WIFI-MESH itself can be initialized.

```
ESP_ERROR_CHECK(esp_netif_init());

/*  event initialization */
ESP_ERROR_CHECK(esp_event_loop_create_default());

/*  Wi-Fi initialization */
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
ESP_ERROR_CHECK(esp_wifi_init(&config));
/*  register IP events handler */
ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &ip_
↪event_handler, NULL));
ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
ESP_ERROR_CHECK(esp_wifi_start());
```

After initializing LwIP and Wi-Fi, the process of getting an ESP-WIFI-MESH network up and running can be summarized into the following three steps:

1. *Initialize Mesh*
2. *Configuring an ESP-WIFI-MESH Network*
3. *Start Mesh*

Initialize Mesh The following code snippet demonstrates how to initialize ESP-WIFI-MESH

```
/*  mesh initialization */
ESP_ERROR_CHECK(esp_mesh_init());
/*  register mesh events handler */
ESP_ERROR_CHECK(esp_event_handler_register(MESH_EVENT, ESP_EVENT_ANY_ID, &mesh_
↪event_handler, NULL));
```

Configuring an ESP-WIFI-MESH Network ESP-WIFI-MESH is configured via *esp_mesh_set_config()* which receives its arguments using the *mesh_cfg_t* structure. The structure contains the following parameters used to configure ESP-WIFI-MESH:

Parameter	Description
Channel	Range from 1 to 14
Mesh ID	ID of ESP-WIFI-MESH Network, see *mesh_addr_t*
Router	Router Configuration, see *mesh_router_t*
Mesh AP	Mesh AP Configuration, see *mesh_ap_cfg_t*
Crypto Functions	Crypto Functions for Mesh IE, see `mesh_crypto_funcs_t`

The following code snippet demonstrates how to configure ESP-WIFI-MESH.

```
/* Enable the Mesh IE encryption by default */
mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT();
/* mesh ID */
memcpy((uint8_t *) &cfg.mesh_id, MESH_ID, 6);
/* channel (must match the router's channel) */
cfg.channel = CONFIG_MESH_CHANNEL;
/* router */
cfg.router.ssid_len = strlen(CONFIG_MESH_ROUTER_SSID);
memcpy((uint8_t *) &cfg.router.ssid, CONFIG_MESH_ROUTER_SSID, cfg.router.ssid_len);
memcpy((uint8_t *) &cfg.router.password, CONFIG_MESH_ROUTER_PASSWD,
       strlen(CONFIG_MESH_ROUTER_PASSWD));
/* mesh softAP */
cfg.mesh_ap.max_connection = CONFIG_MESH_AP_CONNECTIONS;
memcpy((uint8_t *) &cfg.mesh_ap.password, CONFIG_MESH_AP_PASSWD,
       strlen(CONFIG_MESH_AP_PASSWD));
ESP_ERROR_CHECK(esp_mesh_set_config(&cfg));
```

Start Mesh The following code snippet demonstrates how to start ESP-WIFI-MESH.

```
/* mesh start */
ESP_ERROR_CHECK(esp_mesh_start());
```

After starting ESP-WIFI-MESH, the application should check for ESP-WIFI-MESH events to determine when it has connected to the network. After connecting, the application can start transmitting and receiving packets over the ESP-WIFI-MESH network using *esp_mesh_send()* and *esp_mesh_recv()*.

Self Organized Networking Self organized networking is a feature of ESP-WIFI-MESH where nodes can autonomously scan/select/connect/reconnect to other nodes and routers. This feature allows an ESP-WIFI-MESH network to operate with high degree of autonomy by making the network robust to dynamic network topologies and

conditions. With self organized networking enabled, nodes in an ESP-WIFI-MESH network are able to carry out the following actions without autonomously:

- Selection or election of the root node (see **Automatic Root Node Selection** in *Building a Network*)
- Selection of a preferred parent node (see **Parent Node Selection** in *Building a Network*)
- Automatic reconnection upon detecting a disconnection (see **Intermediate Parent Node Failure** in *Managing a Network*)

When self organized networking is enabled, the ESP-WIFI-MESH stack will internally make calls to Wi-Fi APIs. Therefore, **the application layer should not make any calls to Wi-Fi APIs whilst self organized networking is enabled as doing so would risk interfering with ESP-WIFI-MESH**.

Toggling Self Organized Networking Self organized networking can be enabled or disabled by the application at runtime by calling the `esp_mesh_set_self_organized()` function. The function has the two following parameters:

- `bool enable` specifies whether to enable or disable self organized networking.
- `bool select_parent` specifies whether a new parent node should be selected when enabling self organized networking. Selecting a new parent has different effects depending the node type and the node's current state. This parameter is unused when disabling self organized networking.

Disabling Self Organized Networking The following code snippet demonstrates how to disable self organized networking.

```
//Disable self organized networking
esp_mesh_set_self_organized(false, false);
```

ESP-WIFI-MESH will attempt to maintain the node's current Wi-Fi state when disabling self organized networking.

- If the node was previously connected to other nodes, it will remain connected.
- If the node was previously disconnected and was scanning for a parent node or router, it will stop scanning.
- If the node was previously attempting to reconnect to a parent node or router, it will stop reconnecting.

Enabling Self Organized Networking ESP-WIFI-MESH will attempt to maintain the node's current Wi-Fi state when enabling self organized networking. However, depending on the node type and whether a new parent is selected, the Wi-Fi state of the node can change. The following table shows effects of enabling self organized networking.

Select Parent	Is Root Node	Effects
N	N	• Nodes already connected to a parent node will remain connected. • Nodes previously scanning for a parent nodes will stop scanning. Call *esp_mesh_connect()* to restart.
	Y	• A root node already connected to router will stay connected. • A root node disconnected from router will need to call *esp_mesh_connect()* to reconnect.
Y	N	• Nodes without a parent node will automatically select a preferred parent and connect. • Nodes already connected to a parent node will disconnect, reselect a preferred parent node, and connect.
	Y	• For a root node to connect to a parent node, it must give up it's role as root. Therefore, a root node will disconnect from the router and all child nodes, select a preferred parent node, and connect.

The following code snipping demonstrates how to enable self organized networking.

```
//Enable self organized networking and select a new parent
esp_mesh_set_self_organized(true, true);

...

//Enable self organized networking and manually reconnect
esp_mesh_set_self_organized(true, false);
esp_mesh_connect();
```

Calling Wi-Fi API There can be instances in which an application may want to directly call Wi-Fi API whilst using ESP-WIFI-MESH. For example, an application may want to manually scan for neighboring APs. However, **self organized networking must be disabled before the application calls any Wi-Fi APIs**. This will prevent the ESP-WIFI-MESH stack from attempting to call any Wi-Fi APIs and potentially interfering with the application's calls.

Therefore, application calls to Wi-Fi APIs should be placed in between calls of *esp_mesh_set_self_organized()* which disable and enable self organized networking. The following code snippet demonstrates how an application can safely call *esp_wifi_scan_start()* whilst using

ESP-WIFI-MESH.

```
//Disable self organized networking
esp_mesh_set_self_organized(0, 0);

//Stop any scans already in progress
esp_wifi_scan_stop();
//Manually start scan. Will automatically stop when run to completion
esp_wifi_scan_start();

//Process scan results

...

//Re-enable self organized networking if still connected
esp_mesh_set_self_organized(1, 0);

...

//Re-enable self organized networking if non-root and disconnected
esp_mesh_set_self_organized(1, 1);

...

//Re-enable self organized networking if root and disconnected
esp_mesh_set_self_organized(1, 0);  //Don't select new parent
esp_mesh_connect();                 //Manually reconnect to router
```

Application Examples ESP-IDF contains these ESP-WIFI-MESH example projects:

The Internal Communication Example demonstrates how to set up a ESP-WIFI-MESH network and have the root node send a data packet to every node within the network.

The Manual Networking Example demonstrates how to use ESP-WIFI-MESH without the self-organizing features. This example shows how to program a node to manually scan for a list of potential parent nodes and select a parent node based on custom criteria.

API Reference

Header File

- components/esp_wifi/include/esp_mesh.h

Functions

esp_err_t **esp_mesh_init**(void)

Mesh initialization.

- Check whether Wi-Fi is started.
- Initialize mesh global variables with default values.

Attention This API shall be called after Wi-Fi is started.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_deinit**(void)

Mesh de-initialization.

```
        - Release resources and stop the mesh
```

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_start**(void)

Start mesh.

- Initialize mesh IE.
- Start mesh network management service.
- Create TX and RX queues according to the configuration.
- Register mesh packets receive callback.

Attention This API shall be called after mesh initialization and configuration.

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_NOT_INIT
- ESP_ERR_MESH_NOT_CONFIG
- ESP_ERR_MESH_NO_MEMORY

esp_err_t **esp_mesh_stop**(void)

Stop mesh.

- Deinitialize mesh IE.
- Disconnect with current parent.
- Disassociate all currently associated children.
- Stop mesh network management service.
- Unregister mesh packets receive callback.
- Delete TX and RX queues.
- Release resources.
- Restore Wi-Fi softAP to default settings if Wi-Fi dual mode is enabled.
- Set Wi-Fi Power Save type to WIFI_PS_NONE.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_send**(const *mesh_addr_t* *to, const *mesh_data_t* *data, int flag, const *mesh_opt_t* opt[], int opt_count)

Send a packet over the mesh network.

- Send a packet to any device in the mesh network.
- Send a packet to external IP network.

Attention This API is not reentrant.

Parameters

- **`to`** –**[in]** the address of the final destination of the packet
  - If the packet is to the root, set this parameter to NULL.
  - If the packet is to an external IP network, set this parameter to the IPv4:PORT combination. This packet will be delivered to the root firstly, then the root will forward this packet to the final IP server address.
- **`data`** –**[in]** pointer to a sending mesh packet
  - Field size should not exceed MESH_MPS. Note that the size of one mesh packet should not exceed MESH_MTU.
  - Field proto should be set to data protocol in use (default is MESH_PROTO_BIN for binary).
  - Field tos should be set to transmission tos (type of service) in use (default is MESH_TOS_P2P for point-to-point reliable).
- **`flag`** –**[in]** bitmap for data sent
  - Speed up the route search
    - If the packet is to the root and "to" parameter is NULL, set this parameter to 0.
    - If the packet is to an internal device, MESH_DATA_P2P should be set.
    - If the packet is to the root ("to" parameter isn't NULL) or to external IP network, MESH_DATA_TODS should be set.
    - If the packet is from the root to an internal device, MESH_DATA_FROMDS should be set.
  - Specify whether this API is block or non-block, block by default
    - If needs non-blocking, MESH_DATA_NONBLOCK should be set. Otherwise, may use esp_mesh_send_block_time() to specify a blocking time.
  - In the situation of the root change, MESH_DATA_DROP identifies this packet can be dropped by the new root for upstream data to external IP network, we try our best to avoid data loss caused by the root change, but there is a risk that the new root is running out of memory because most of memory is occupied by the pending data which isn't read out in time by esp_mesh_recv_toDS().
    Generally, we suggest esp_mesh_recv_toDS() is called after a connection with IP network is created. Thus data outgoing to external IP network via socket is just from reading esp_mesh_recv_toDS() which avoids unnecessary memory copy.
- **`opt`** –**[in]** options
  - In case of sending a packet to a certain group, MESH_OPT_SEND_GROUP is a good choice. In this option, the value field should be set to the target receiver addresses in this group.
  - Root sends a packet to an internal device, this packet is from external IP network in case the receiver device responds this packet, MESH_OPT_RECV_DS_ADDR is required to attach the target DS address.
- **`opt_count`** –**[in]** option count
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_DISCONNECTED
- ESP_ERR_MESH_OPT_UNKNOWN
- ESP_ERR_MESH_EXCEED_MTU
- ESP_ERR_MESH_NO_MEMORY
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_NO_ROUTE_FOUND
- ESP_ERR_MESH_DISCARD

esp_err_t **`esp_mesh_send_block_time`**(uint32_t time_ms)

Set blocking time of esp_mesh_send()

Attention This API shall be called before mesh is started.

Parameters **`time_ms`** –**[in]** blocking time of esp_mesh_send(), unit:ms
Returns
- ESP_OK

esp_err_t **`esp_mesh_recv`**(*mesh_addr_t* *from, *mesh_data_t* *data, int timeout_ms, int *flag, *mesh_opt_t* opt[], int opt_count)

Receive a packet targeted to self over the mesh network.

flag could be MESH_DATA_FROMDS or MESH_DATA_TODS.

Attention Mesh RX queue should be checked regularly to avoid running out of memory.
- Use esp_mesh_get_rx_pending() to check the number of packets available in the queue waiting to be received by applications.

Parameters
- **`from`** –**[out]** the address of the original source of the packet
- **`data`** –**[out]** pointer to the received mesh packet
  - Field proto is the data protocol in use. Should follow it to parse the received data.
  - Field tos is the transmission tos (type of service) in use.
- **`timeout_ms`** –**[in]** wait time if a packet isn't immediately available (0:no wait, portMAX_DELAY:wait forever)
- **`flag`** –**[out]** bitmap for data received
  - MESH_DATA_FROMDS represents data from external IP network
  - MESH_DATA_TODS represents data directed upward within the mesh network
- **`opt`** –**[out]** options desired to receive
  - MESH_OPT_RECV_DS_ADDR attaches the DS address
- **`opt_count`** –**[in]** option count desired to receive
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_DISCARD

esp_err_t **`esp_mesh_recv_toDS`**(*mesh_addr_t* *from, *mesh_addr_t* *to, *mesh_data_t* *data, int timeout_ms, int *flag, *mesh_opt_t* opt[], int opt_count)

Receive a packet targeted to external IP network.

- Root uses this API to receive packets destined to external IP network
- Root forwards the received packets to the final destination via socket.
- If no socket connection is ready to send out the received packets and this esp_mesh_recv_toDS() hasn't been called by applications, packets from the whole mesh network will be pending in toDS queue.

Use esp_mesh_get_rx_pending() to check the number of packets available in the queue waiting to be received by applications in case of running out of memory in the root.

Using esp_mesh_set_xon_qsize() users may configure the RX queue size, default:32. If this size is too large, and esp_mesh_recv_toDS() isn't called in time, there is a risk that a great deal of memory is occupied by the pending packets. If this size is too small, it will impact the efficiency on upstream. How to decide this value depends on the specific application scenarios.

flag could be MESH_DATA_TODS.

Attention This API is only called by the root.

Parameters

- **`from`** –**[out]** the address of the original source of the packet
- **`to`** –**[out]** the address contains remote IP address and port (IPv4:PORT)
- **`data`** –**[out]** pointer to the received packet
  - Contain the protocol and applications should follow it to parse the data.
- **`timeout_ms`** –**[in]** wait time if a packet isn't immediately available (0:no wait, portMAX_DELAY:wait forever)
- **`flag`** –**[out]** bitmap for data received
  - MESH_DATA_TODS represents the received data target to external IP network. Root shall forward this data to external IP network via the association with router.
- **`opt`** –**[out]** options desired to receive
- **`opt_count`** –**[in]** option count desired to receive

Returns

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_DISCARD
- ESP_ERR_MESH_RECV_RELEASE

esp_err_t **`esp_mesh_set_config`**(const *mesh_cfg_t* *config)

Set mesh stack configuration.

- Use MESH_INIT_CONFIG_DEFAULT() to initialize the default values, mesh IE is encrypted by default.
- Mesh network is established on a fixed channel (1-14).
- Mesh event callback is mandatory.
- Mesh ID is an identifier of an MBSS. Nodes with the same mesh ID can communicate with each other.
- Regarding to the router configuration, if the router is hidden, BSSID field is mandatory.

If BSSID field isn't set and there exists more than one router with same SSID, there is a risk that more roots than one connected with different BSSID will appear. It means more than one mesh network is established with the same mesh ID.

Root conflict function could eliminate redundant roots connected with the same BSSID, but couldn't handle roots connected with different BSSID. Because users might have such requirements of setting up routers with same SSID for the future replacement. But in that case, if the above situations happen, please make sure applications implement forward functions on the root to guarantee devices in different mesh networks can communicate with each other. max_connection of mesh softAP is limited by the max number of Wi-Fi softAP supported (max:10).

Attention This API shall be called before mesh is started after mesh is initialized.

Parameters **`config`** –**[in]** pointer to mesh stack configuration

Returns

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

esp_err_t **`esp_mesh_get_config`**(*mesh_cfg_t* *config)

Get mesh stack configuration.

Parameters `config` –**[out]** pointer to mesh stack configuration
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_set_router**(const *mesh_router_t* *router)

Get router configuration.

Attention This API is used to dynamically modify the router configuration after mesh is configured.

Parameters `router` –**[in]** pointer to router configuration
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_get_router**(*mesh_router_t* *router)

Get router configuration.

Parameters `router` –**[out]** pointer to router configuration
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_set_id**(const *mesh_addr_t* *id)

Set mesh network ID.

Attention This API is used to dynamically modify the mesh network ID.

Parameters `id` –**[in]** pointer to mesh network ID
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT: invalid argument

esp_err_t **esp_mesh_get_id**(*mesh_addr_t* *id)

Get mesh network ID.

Parameters `id` –**[out]** pointer to mesh network ID
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_set_type**(*mesh_type_t* type)

Designate device type over the mesh network.

- MESH_IDLE: designates a device as a self-organized node for a mesh network
- MESH_ROOT: designates the root node for a mesh network
- MESH_LEAF: designates a device as a standalone Wi-Fi station that connects to a parent
- MESH_STA: designates a device as a standalone Wi-Fi station that connects to a router

Parameters `type` –**[in]** device type
Returns
- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED

mesh_type_t **esp_mesh_get_type**(void)

Get device type over mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Returns mesh type

esp_err_t **esp_mesh_set_max_layer**(int max_layer)

Set network max layer value.

- for tree topology, the max is 25.
- for chain topology, the max is 1000.
- Network max layer limits the max hop count.

Attention This API shall be called before mesh is started.

Parameters **max_layer** –**[in]** max layer value

Returns

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

int **esp_mesh_get_max_layer**(void)

Get max layer value.

Returns max layer value

esp_err_t **esp_mesh_set_ap_password**(const uint8_t *pwd, int len)

Set mesh softAP password.

Attention This API shall be called before mesh is started.

Parameters

- **pwd** –**[in]** pointer to the password
- **len** –**[in]** password length

Returns

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

esp_err_t **esp_mesh_set_ap_authmode**(*wifi_auth_mode_t* authmode)

Set mesh softAP authentication mode.

Attention This API shall be called before mesh is started.

Parameters **authmode** –**[in]** authentication mode

Returns

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

wifi_auth_mode_t **esp_mesh_get_ap_authmode**(void)

Get mesh softAP authentication mode.

Returns authentication mode

esp_err_t **esp_mesh_set_ap_connections**(int connections)

Set mesh max connection value.

- Set mesh softAP max connection = mesh max connection + non-mesh max connection

Attention This API shall be called before mesh is started.

Parameters **connections** –**[in]** the number of max connections
Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

int **esp_mesh_get_ap_connections**(void)

Get mesh max connection configuration.

Returns the number of mesh max connections

int **esp_mesh_get_non_mesh_connections**(void)

Get non-mesh max connection configuration.

Returns the number of non-mesh max connections

int **esp_mesh_get_layer**(void)

Get current layer value over the mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Returns layer value

esp_err_t **esp_mesh_get_parent_bssid**(*mesh_addr_t* *bssid)

Get the parent BSSID.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Parameters **bssid** –**[out]** pointer to parent BSSID
Returns
- ESP_OK
- ESP_FAIL

bool **esp_mesh_is_root**(void)

Return whether the device is the root node of the network.

Returns true/false

esp_err_t **esp_mesh_set_self_organized**(bool enable, bool select_parent)

Enable/disable self-organized networking.

- Self-organized networking has three main functions: select the root node; find a preferred parent; initiate reconnection if a disconnection is detected.
- Self-organized networking is enabled by default.
- If self-organized is disabled, users should set a parent for the device via esp_mesh_set_parent().

Attention This API is used to dynamically modify whether to enable the self organizing.

Parameters
- **`enable`** –**[in]** enable or disable self-organized networking
- **`select_parent`** –**[in]** Only valid when self-organized networking is enabled.
  - if select_parent is set to true, the root will give up its mesh root status and search for a new parent like other non-root devices.

Returns
- ESP_OK
- ESP_FAIL

bool **`esp_mesh_get_self_organized`**(void)

Return whether enable self-organized networking or not.

Returns true/false

esp_err_t **`esp_mesh_waive_root`**(const *mesh_vote_t* *vote, int reason)

Cause the root device to give up (waive) its mesh root status.

- A device is elected root primarily based on RSSI from the external router.
- If external router conditions change, users can call this API to perform a root switch.
- In this API, users could specify a desired root address to replace itself or specify an attempts value to ask current root to initiate a new round of voting. During the voting, a better root candidate would be expected to find to replace the current one.
- If no desired root candidate, the vote will try a specified number of attempts (at least 15). If no better root candidate is found, keep the current one. If a better candidate is found, the new better one will send a root switch request to the current root, current root will respond with a root switch acknowledgment.
- After that, the new candidate will connect to the router to be a new root, the previous root will disconnect with the router and choose another parent instead.

Root switch is completed with minimal disruption to the whole mesh network.

Attention This API is only called by the root.

Parameters
- **`vote`** –**[in]** vote configuration
  - If this parameter is set NULL, the vote will perform the default 15 times.
  - Field percentage threshold is 0.9 by default.
  - Field is_rc_specified shall be false.
  - Field attempts shall be at least 15 times.
- **`reason`** –**[in]** only accept MESH_VOTE_REASON_ROOT_INITIATED for now

Returns
- ESP_OK
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_DISCARD
- ESP_FAIL

esp_err_t **`esp_mesh_set_vote_percentage`**(float percentage)

Set vote percentage threshold for approval of being a root (default:0.9)

- During the networking, only obtaining vote percentage reaches this threshold, the device could be a root.

Attention This API shall be called before mesh is started.

Parameters **`percentage`** –**[in]** vote percentage threshold
Returns
- ESP_OK
- ESP_FAIL

float **`esp_mesh_get_vote_percentage`**(void)

Get vote percentage threshold for approval of being a root.

Returns percentage threshold

esp_err_t **`esp_mesh_set_ap_assoc_expire`**(int seconds)

Set mesh softAP associate expired time (default:10 seconds)

- If mesh softAP hasn't received any data from an associated child within this time, mesh softAP will take this child inactive and disassociate it.
- If mesh softAP is encrypted, this value should be set a greater value, such as 30 seconds.

Parameters **`seconds`** –**[in]** the expired time
Returns
- ESP_OK
- ESP_FAIL

int **`esp_mesh_get_ap_assoc_expire`**(void)

Get mesh softAP associate expired time.

Returns seconds

int **`esp_mesh_get_total_node_num`**(void)

Get total number of devices in current network (including the root)

Attention The returned value might be incorrect when the network is changing.

Returns total number of devices (including the root)

int **`esp_mesh_get_routing_table_size`**(void)

Get the number of devices in this device's sub-network (including self)

Returns the number of devices over this device's sub-network (including self)

esp_err_t **`esp_mesh_get_routing_table`**(*mesh_addr_t* *mac, int len, int *size)

Get routing table of this device's sub-network (including itself)

Parameters
- **`mac`** –**[out]** pointer to routing table
- **`len`** –**[in]** routing table size(in bytes)
- **`size`** –**[out]** pointer to the number of devices in routing table (including itself)

Returns
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

esp_err_t **`esp_mesh_post_toDS_state`**(bool reachable)

Post the toDS state to the mesh stack.

Attention This API is only for the root.

Parameters **reachable** –**[in]** this state represents whether the root is able to access external IP network

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_get_tx_pending**(*mesh_tx_pending_t* *pending)

Return the number of packets pending in the queue waiting to be sent by the mesh stack.

Parameters **pending** –**[out]** pointer to the TX pending

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_get_rx_pending**(*mesh_rx_pending_t* *pending)

Return the number of packets available in the queue waiting to be received by applications.

Parameters **pending** –**[out]** pointer to the RX pending

Returns

- ESP_OK
- ESP_FAIL

int **esp_mesh_available_txupQ_num**(const *mesh_addr_t* *addr, uint32_t *xseqno_in)

Return the number of packets could be accepted from the specified address.

Parameters

- **addr** –**[in]** self address or an associate children address
- **xseqno_in** –**[out]** sequence number of the last received packet from the specified address

Returns the number of upQ for a certain address

esp_err_t **esp_mesh_set_xon_qsize**(int qsize)

Set the number of RX queue for the node, the average number of window allocated to one of its child node is: wnd = xon_qsize / (2 * max_connection + 1). However, the window of each child node is not strictly equal to the average value, it is affected by the traffic also.

Attention This API shall be called before mesh is started.

Parameters **qsize** –**[in]** default:32 (min:16)

Returns

- ESP_OK
- ESP_FAIL

int **esp_mesh_get_xon_qsize**(void)

Get queue size.

Returns the number of queue

esp_err_t **esp_mesh_allow_root_conflicts**(bool allowed)

Set whether allow more than one root existing in one network.

Parameters **allowed** –**[in]** allow or not

Returns

- ESP_OK
- ESP_WIFI_ERR_NOT_INIT
- ESP_WIFI_ERR_NOT_START

bool **esp_mesh_is_root_conflicts_allowed**(void)

Check whether allow more than one root to exist in one network.

Returns true/false

esp_err_t **esp_mesh_set_group_id**(const *mesh_addr_t* *addr, int num)

Set group ID addresses.

Parameters
- **addr** –**[in]** pointer to new group ID addresses
- **num** –**[in]** the number of group ID addresses

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

esp_err_t **esp_mesh_delete_group_id**(const *mesh_addr_t* *addr, int num)

Delete group ID addresses.

Parameters
- **addr** –**[in]** pointer to deleted group ID address
- **num** –**[in]** the number of group ID addresses

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

int **esp_mesh_get_group_num**(void)

Get the number of group ID addresses.

Returns the number of group ID addresses

esp_err_t **esp_mesh_get_group_list**(*mesh_addr_t* *addr, int num)

Get group ID addresses.

Parameters
- **addr** –**[out]** pointer to group ID addresses
- **num** –**[in]** the number of group ID addresses

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

bool **esp_mesh_is_my_group**(const *mesh_addr_t* *addr)

Check whether the specified group address is my group.

Returns true/false

esp_err_t **esp_mesh_set_capacity_num**(int num)

Set mesh network capacity (max:1000, default:300)

Attention This API shall be called before mesh is started.

Parameters **num** –**[in]** mesh network capacity

Returns
- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED
- ESP_MESH_ERR_ARGUMENT

int **esp_mesh_get_capacity_num**(void)

Get mesh network capacity.

Returns mesh network capacity

esp_err_t **esp_mesh_set_ie_crypto_funcs**(const mesh_crypto_funcs_t *crypto_funcs)

Set mesh IE crypto functions.

Attention This API can be called at any time after mesh is initialized.

Parameters **crypto_funcs** –**[in]** crypto functions for mesh IE
- If crypto_funcs is set to NULL, mesh IE is no longer encrypted.

Returns
- ESP_OK

esp_err_t **esp_mesh_set_ie_crypto_key**(const char *key, int len)

Set mesh IE crypto key.

Attention This API can be called at any time after mesh is initialized.

Parameters
- **key** –**[in]** ASCII crypto key
- **len** –**[in]** length in bytes, range:8~64

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

esp_err_t **esp_mesh_get_ie_crypto_key**(char *key, int len)

Get mesh IE crypto key.

Parameters
- **key** –**[out]** ASCII crypto key
- **len** –**[in]** length in bytes, range:8~64

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

esp_err_t **esp_mesh_set_root_healing_delay**(int delay_ms)

Set delay time before starting root healing.

Parameters **delay_ms** –**[in]** delay time in milliseconds

Returns
- ESP_OK

int **esp_mesh_get_root_healing_delay**(void)

Get delay time before network starts root healing.

Returns delay time in milliseconds

esp_err_t **esp_mesh_fix_root**(bool enable)

Enable network Fixed Root Setting.

- Enabling fixed root disables automatic election of the root node via voting.
- All devices in the network shall use the same Fixed Root Setting (enabled or disabled).
- If Fixed Root is enabled, users should make sure a root node is designated for the network.

Parameters **enable** –**[in]** enable or not

Returns
- ESP_OK

bool **esp_mesh_is_root_fixed**(void)

Check whether network Fixed Root Setting is enabled.

- Enable/disable network Fixed Root Setting by API esp_mesh_fix_root().
- Network Fixed Root Setting also changes with the "flag" value in parent networking IE.

Returns true/false

esp_err_t **esp_mesh_set_parent**(const *wifi_config_t* *parent, const *mesh_addr_t* *parent_mesh_id, *mesh_type_t* my_type, int my_layer)

Set a specified parent for the device.

Attention This API can be called at any time after mesh is configured.

Parameters

- **parent** –**[in]** parent configuration, the SSID and the channel of the parent are mandatory.
  - If the BSSID is set, make sure that the SSID and BSSID represent the same parent, otherwise the device will never find this specified parent.
- **parent_mesh_id** –**[in]** parent mesh ID,
  - If this value is not set, the original mesh ID is used.
- **my_type** –**[in]** mesh type
  - MESH_STA is not supported.
  - If the parent set for the device is the same as the router in the network configuration, then my_type shall set MESH_ROOT and my_layer shall set MESH_ROOT_LAYER.
- **my_layer** –**[in]** mesh layer
  - my_layer of the device may change after joining the network.
  - If my_type is set MESH_NODE, my_layer shall be greater than MESH_ROOT_LAYER.
  - If my_type is set MESH_LEAF, the device becomes a standalone Wi-Fi station and no longer has the ability to extend the network.

Returns

- ESP_OK
- ESP_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_CONFIG

esp_err_t **esp_mesh_scan_get_ap_ie_len**(int *len)

Get mesh networking IE length of one AP.

Parameters **len** –**[out]** mesh networking IE length

Returns

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG
- ESP_ERR_WIFI_FAIL

esp_err_t **esp_mesh_scan_get_ap_record**(*wifi_ap_record_t* *ap_record, void *buffer)

Get AP record.

Attention Different from esp_wifi_scan_get_ap_records(), this API only gets one of APs scanned each time. See "manual_networking" example.

Parameters

- **ap_record** –**[out]** pointer to one AP record
- **buffer** –**[out]** pointer to the mesh networking IE of this AP

Returns

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG
- ESP_ERR_WIFI_FAIL

esp_err_t **esp_mesh_flush_upstream_packets**(void)

Flush upstream packets pending in to_parent queue and to_parent_p2p queue.

Returns

- ESP_OK

esp_err_t **esp_mesh_get_subnet_nodes_num**(const *mesh_addr_t* *child_mac, int *nodes_num)

Get the number of nodes in the subnet of a specific child.

Parameters

- **child_mac** –**[in]** an associated child address of this device
- **nodes_num** –**[out]** pointer to the number of nodes in the subnet of a specific child

Returns

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_get_subnet_nodes_list**(const *mesh_addr_t* *child_mac, *mesh_addr_t* *nodes, int nodes_num)

Get nodes in the subnet of a specific child.

Parameters

- **child_mac** –**[in]** an associated child address of this device
- **nodes** –**[out]** pointer to nodes in the subnet of a specific child
- **nodes_num** –**[in]** the number of nodes in the subnet of a specific child

Returns

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

esp_err_t **esp_mesh_disconnect**(void)

Disconnect from current parent.

Returns

- ESP_OK

esp_err_t **esp_mesh_connect**(void)

Connect to current parent.

Returns

- ESP_OK

esp_err_t **esp_mesh_flush_scan_result**(void)

Flush scan result.

Returns

- ESP_OK

esp_err_t **esp_mesh_switch_channel**(const uint8_t *new_bssid, int csa_newchan, int csa_count)

Cause the root device to add Channel Switch Announcement Element (CSA IE) to beacon.

- Set the new channel
- Set how many beacons with CSA IE will be sent before changing a new channel
- Enable the channel switch function

Attention This API is only called by the root.

Parameters
- **new_bssid** –**[in]** the new router BSSID if the router changes
- **csa_newchan** –**[in]** the new channel number to which the whole network is moving
- **csa_count** –**[in]** channel switch period(beacon count), unit is based on beacon interval of its softAP, the default value is 15.

Returns
- ESP_OK

esp_err_t **esp_mesh_get_router_bssid**(uint8_t *router_bssid)

Get the router BSSID.

Parameters **router_bssid** –**[out]** pointer to the router BSSID

Returns
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG

int64_t **esp_mesh_get_tsf_time**(void)

Get the TSF time.

Returns the TSF time

esp_err_t **esp_mesh_set_topology**(*esp_mesh_topology_t* topo)

Set mesh topology. The default value is MESH_TOPO_TREE.

- MESH_TOPO_CHAIN supports up to 1000 layers

Attention This API shall be called before mesh is started.

Parameters **topo** –**[in]** MESH_TOPO_TREE or MESH_TOPO_CHAIN

Returns
- ESP_OK
- ESP_MESH_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

esp_mesh_topology_t **esp_mesh_get_topology**(void)

Get mesh topology.

Returns MESH_TOPO_TREE or MESH_TOPO_CHAIN

esp_err_t **esp_mesh_enable_ps**(void)

Enable mesh Power Save function.

Attention This API shall be called before mesh is started.

Returns
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_MESH_NOT_ALLOWED

esp_err_t **esp_mesh_disable_ps**(void)

Disable mesh Power Save function.

Attention This API shall be called before mesh is started.

Returns
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_MESH_NOT_ALLOWED

bool **esp_mesh_is_ps_enabled**(void)

Check whether the mesh Power Save function is enabled.

Returns true/false

bool **esp_mesh_is_device_active**(void)

Check whether the device is in active state.

- If the device is not in active state, it will neither transmit nor receive frames.

Returns true/false

esp_err_t **esp_mesh_set_active_duty_cycle**(int dev_duty, int dev_duty_type)

Set the device duty cycle and type.

- The range of dev_duty values is 1 to 100. The default value is 10.
- dev_duty = 100, the PS will be stopped.
- dev_duty is better to not less than 5.
- dev_duty_type could be MESH_PS_DEVICE_DUTY_REQUEST or MESH_PS_DEVICE_DUTY_DEMAND.
- If dev_duty_type is set to MESH_PS_DEVICE_DUTY_REQUEST, the device will use a nwk_duty provided by the network.
- If dev_duty_type is set to MESH_PS_DEVICE_DUTY_DEMAND, the device will use the specified dev_duty.

Attention This API can be called at any time after mesh is started.

Parameters
- **dev_duty** –**[in]** device duty cycle
- **dev_duty_type** –**[in]** device PS duty cycle type, not accept MESH_PS_NETWORK_DUTY_MASTER

Returns
- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_get_active_duty_cycle**(int *dev_duty, int *dev_duty_type)

Get device duty cycle and type.

Parameters
- **dev_duty** –**[out]** device duty cycle
- **dev_duty_type** –**[out]** device PS duty cycle type

Returns
- ESP_OK

esp_err_t **esp_mesh_set_network_duty_cycle**(int nwk_duty, int duration_mins, int applied_rule)

Set the network duty cycle, duration and rule.

- The range of nwk_duty values is 1 to 100. The default value is 10.
- nwk_duty is the network duty cycle the entire network or the up-link path will use. A device that successfully sets the nwk_duty is known as a NWK-DUTY-MASTER.

- duration_mins specifies how long the specified nwk_duty will be used. Once duration_mins expires, the root will take over as the NWK-DUTY-MASTER. If an existing NWK-DUTY-MASTER leaves the network, the root will take over as the NWK-DUTY-MASTER again.
- duration_mins = (-1) represents nwk_duty will be used until a new NWK-DUTY-MASTER with a different nwk_duty appears.
- Only the root can set duration_mins to (-1).
- If applied_rule is set to MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE, the nwk_duty will be used by the entire network.
- If applied_rule is set to MESH_PS_NETWORK_DUTY_APPLIED_UPLINK, the nwk_duty will only be used by the up-link path nodes.
- The root does not accept MESH_PS_NETWORK_DUTY_APPLIED_UPLINK.
- A nwk_duty with duration_mins(-1) set by the root is the default network duty cycle used by the entire network.

Attention This API can be called at any time after mesh is started.

- In self-organized network, if this API is called before mesh is started in all devices, (1)nwk_duty shall be set to the same value for all devices; (2)duration_mins shall be set to (-1); (3)applied_rule shall be set to MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE; after the voted root appears, the root will become the NWK-DUTY-MASTER and broadcast the nwk_duty and its identity of NWK-DUTY-MASTER.
- If the root is specified (FIXED-ROOT), call this API in the root to provide a default nwk_duty for the entire network.
- After joins the network, any device can call this API to change the nwk_duty, duration_mins or applied_rule.

Parameters

- **`nwk_duty`** –**[in]** network duty cycle
- **`duration_mins`** –**[in]** duration (unit: minutes)
- **`applied_rule`** –**[in]** only support MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **`esp_mesh_get_network_duty_cycle`**(int *nwk_duty, int *duration_mins, int *dev_duty_type, int *applied_rule)

Get the network duty cycle, duration, type and rule.

Parameters

- **`nwk_duty`** –**[out]** current network duty cycle
- **`duration_mins`** –**[out]** the duration of current nwk_duty
- **`dev_duty_type`** –**[out]** if it includes MESH_PS_DEVICE_DUTY_MASTER, this device is the current NWK-DUTY-MASTER.
- **`applied_rule`** –**[out]** MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE

Returns

- ESP_OK

int **`esp_mesh_get_running_active_duty_cycle`**(void)

Get the running active duty cycle.

- The running active duty cycle of the root is 100.
- If duty type is set to MESH_PS_DEVICE_DUTY_REQUEST, the running active duty cycle is nwk_duty provided by the network.
- If duty type is set to MESH_PS_DEVICE_DUTY_DEMAND, the running active duty cycle is dev_duty specified by the users.
- In a mesh network, devices are typically working with a certain duty-cycle (transmitting, receiving and sleep) to reduce the power consumption. The running active duty cycle decides the amount of awake time within a beacon interval. At each start of beacon interval, all devices wake up, broadcast beacons,

and transmit packets if they do have pending packets for their parents or for their children. Note that Low-duty-cycle means devices may not be active in most of the time, the latency of data transmission might be greater.

Returns the running active duty cycle

esp_err_t **esp_mesh_ps_duty_signaling**(int fwd_times)

Duty signaling.

Parameters **fwd_times** –**[in]** the times of forwarding duty signaling packets
Returns
- ESP_OK

Unions

union **mesh_addr_t**

#include <esp_mesh.h> Mesh address.

Public Members

uint8_t **addr**[6]

mac address

mip_t **mip**

mip address

union **mesh_event_info_t**

#include <esp_mesh.h> Mesh event information.

Public Members

mesh_event_channel_switch_t **channel_switch**

channel switch

mesh_event_child_connected_t **child_connected**

child connected

mesh_event_child_disconnected_t **child_disconnected**

child disconnected

mesh_event_routing_table_change_t **routing_table**

routing table change

mesh_event_connected_t **connected**

parent connected

mesh_event_disconnected_t **disconnected**

parent disconnected

mesh_event_no_parent_found_t **no_parent**

no parent found

mesh_event_layer_change_t **layer_change**

layer change

mesh_event_toDS_state_t **toDS_state**

toDS state, devices shall check this state firstly before trying to send packets to external IP network. This state indicates right now whether the root is capable of sending packets out. If not, devices had better to wait until this state changes to be MESH_TODS_REACHABLE.

mesh_event_vote_started_t **vote_started**

vote started

mesh_event_root_address_t **root_addr**

root address

mesh_event_root_switch_req_t **switch_req**

root switch request

mesh_event_root_conflict_t **root_conflict**

other powerful root

mesh_event_root_fixed_t **root_fixed**

fixed root

mesh_event_scan_done_t **scan_done**

scan done

mesh_event_network_state_t **network_state**

network state, such as whether current mesh network has a root.

mesh_event_find_network_t **find_network**

network found that can join

mesh_event_router_switch_t **router_switch**

new router information

mesh_event_ps_duty_t **ps_duty**

PS duty information

union **mesh_rc_config_t**

#include <esp_mesh.h> Vote address configuration.

Public Members

int **attempts**

max vote attempts before a new root is elected automatically by mesh network. (min:15, 15 by default)

mesh_addr_t **rc_addr**

a new root address specified by users for API esp_mesh_waive_root()

Structures

struct **mip_t**

IP address and port.

Public Members

ip4_addr_t **ip4**

IP address

uint16_t **port**

port

struct **mesh_event_channel_switch_t**

Channel switch information.

Public Members

uint8_t **channel**

new channel

struct **mesh_event_connected_t**

Parent connected information.

Public Members

wifi_event_sta_connected_t **connected**

parent information, same as Wi-Fi event SYSTEM_EVENT_STA_CONNECTED does

uint16_t **self_layer**

layer

uint8_t **duty**

parent duty

struct **mesh_event_no_parent_found_t**

No parent found information.

Public Members

int **scan_times**

scan times being through

struct **mesh_event_layer_change_t**

Layer change information.

Public Members

uint16_t **new_layer**

new layer

struct **mesh_event_vote_started_t**

vote started information

Public Members

int **reason**

vote reason, vote could be initiated by children or by the root itself

int **attempts**

max vote attempts before stopped

mesh_addr_t **rc_addr**

root address specified by users via API esp_mesh_waive_root()

struct **mesh_event_find_network_t**

find a mesh network that this device can join

Public Members

uint8_t **channel**

channel number of the new found network

uint8_t **router_bssid**[6]

router BSSID

struct **mesh_event_root_switch_req_t**

Root switch request information.

Public Members

int **reason**

root switch reason, generally root switch is initialized by users via API esp_mesh_waive_root()

mesh_addr_t **rc_addr**

the address of root switch requester

struct **mesh_event_root_conflict_t**

Other powerful root address.

Public Members

int8_t **rssi**

rssi with router

uint16_t **capacity**

the number of devices in current network

uint8_t **addr**[6]

other powerful root address

struct **mesh_event_routing_table_change_t**

Routing table change.

Public Members

uint16_t **rt_size_new**

the new value

uint16_t **rt_size_change**

the changed value

struct **mesh_event_root_fixed_t**

Root fixed.

Public Members

bool **is_fixed**

status

struct **mesh_event_scan_done_t**

Scan done event information.

Public Members

uint8_t **number**

the number of APs scanned

struct **mesh_event_network_state_t**

Network state information.

Public Members

bool **is_rootless**

whether current mesh network has a root

struct **mesh_event_ps_duty_t**

PS duty information.

Public Members

uint8_t **duty**

parent or child duty

mesh_event_child_connected_t **child_connected**

child info

struct **mesh_opt_t**

Mesh option.

Public Members

uint8_t **type**

option type

uint16_t **len**

option length

uint8_t ***val**

option value

struct **mesh_data_t**

Mesh data for esp_mesh_send() and esp_mesh_recv()

Public Members

uint8_t ***data**

data

uint16_t **size**

data size

mesh_proto_t **proto**

data protocol

mesh_tos_t **tos**

data type of service

struct **mesh_router_t**

Router configuration.

Public Members

uint8_t **ssid**[32]

SSID

uint8_t **ssid_len**

length of SSID

uint8_t **bssid**[6]

BSSID, if this value is specified, users should also specify "allow_router_switch".

uint8_t **password**[64]

password

bool **allow_router_switch**

if the BSSID is specified and this value is also set, when the router of this specified BSSID fails to be found after "fail" (mesh_attempts_t) times, the whole network is allowed to switch to another router with the same SSID. The new router might also be on a different channel. The default value is false. There is a risk that if the password is different between the new switched router and the previous one, the mesh network could be established but the root will never connect to the new switched router.

struct **mesh_ap_cfg_t**

Mesh softAP configuration.

Public Members

uint8_t **password**[64]

mesh softAP password

uint8_t **max_connection**

max number of stations allowed to connect in, default 6, max 10 = max_connection + non-mesh_max_connection max mesh connections

uint8_t **nonmesh_max_connection**

max non-mesh connections

struct **mesh_cfg_t**

Mesh initialization configuration.

Public Members

uint8_t **channel**

channel, the mesh network on

bool **allow_channel_switch**

if this value is set, when "fail" (mesh_attempts_t) times is reached, device will change to a full channel scan for a network that could join. The default value is false.

mesh_addr_t **mesh_id**

mesh network identification

mesh_router_t **router**

router configuration

mesh_ap_cfg_t **mesh_ap**

mesh softAP configuration

const mesh_crypto_funcs_t ***crypto_funcs**

crypto functions

struct **mesh_vote_t**

Vote.

Public Members

float **percentage**

vote percentage threshold for approval of being a root

bool **is_rc_specified**

if true, rc_addr shall be specified (Unimplemented). if false, attempts value shall be specified to make network start root election.

mesh_rc_config_t **config**

vote address configuration

struct **mesh_tx_pending_t**

The number of packets pending in the queue waiting to be sent by the mesh stack.

Public Members

int **to_parent**

to parent queue

int **to_parent_p2p**

to parent (P2P) queue

int **to_child**

to child queue

int **to_child_p2p**

to child (P2P) queue

int **mgmt**

management queue

int **broadcast**

broadcast and multicast queue

struct **mesh_rx_pending_t**

The number of packets available in the queue waiting to be received by applications.

Public Members

int **toDS**

to external DS

int **toSelf**

to self

Macros

MESH_ROOT_LAYER

root layer value

MESH_MTU

max transmit unit(in bytes)

MESH_MPS

max payload size(in bytes)

ESP_ERR_MESH_WIFI_NOT_START

Mesh error code definition.

Wi-Fi isn't started

ESP_ERR_MESH_NOT_INIT

mesh isn't initialized

ESP_ERR_MESH_NOT_CONFIG

mesh isn't configured

ESP_ERR_MESH_NOT_START

mesh isn't started

ESP_ERR_MESH_NOT_SUPPORT

not supported yet

ESP_ERR_MESH_NOT_ALLOWED

operation is not allowed

ESP_ERR_MESH_NO_MEMORY

out of memory

ESP_ERR_MESH_ARGUMENT
illegal argument

ESP_ERR_MESH_EXCEED_MTU
packet size exceeds MTU

ESP_ERR_MESH_TIMEOUT
timeout

ESP_ERR_MESH_DISCONNECTED
disconnected with parent on station interface

ESP_ERR_MESH_QUEUE_FAIL
queue fail

ESP_ERR_MESH_QUEUE_FULL
queue full

ESP_ERR_MESH_NO_PARENT_FOUND
no parent found to join the mesh network

ESP_ERR_MESH_NO_ROUTE_FOUND
no route found to forward the packet

ESP_ERR_MESH_OPTION_NULL
no option found

ESP_ERR_MESH_OPTION_UNKNOWN
unknown option

ESP_ERR_MESH_XON_NO_WINDOW
no window for software flow control on upstream

ESP_ERR_MESH_INTERFACE
low-level Wi-Fi interface error

ESP_ERR_MESH_DISCARD_DUPLICATE
discard the packet due to the duplicate sequence number

ESP_ERR_MESH_DISCARD
discard the packet

ESP_ERR_MESH_VOTING
vote in progress

ESP_ERR_MESH_XMIT
XMIT

ESP_ERR_MESH_QUEUE_READ

error in reading queue

ESP_ERR_MESH_PS

mesh PS is not specified as enable or disable

ESP_ERR_MESH_RECV_RELEASE

release esp_mesh_recv_toDS

MESH_DATA_ENC

Flags bitmap for esp_mesh_send() and esp_mesh_recv()

data encrypted (Unimplemented)

MESH_DATA_P2P

point-to-point delivery over the mesh network

MESH_DATA_FROMDS

receive from external IP network

MESH_DATA_TODS

identify this packet is target to external IP network

MESH_DATA_NONBLOCK

esp_mesh_send() non-block

MESH_DATA_DROP

in the situation of the root having been changed, identify this packet can be dropped by new root

MESH_DATA_GROUP

identify this packet is target to a group address

MESH_OPT_SEND_GROUP

Option definitions for esp_mesh_send() and esp_mesh_recv()

data transmission by group; used with esp_mesh_send() and shall have payload

MESH_OPT_RECV_DS_ADDR

return a remote IP address; used with esp_mesh_send() and esp_mesh_recv()

MESH_ASSOC_FLAG_VOTE_IN_PROGRESS

Flag of mesh networking IE.

vote in progress

MESH_ASSOC_FLAG_NETWORK_FREE

no root in current network

MESH_ASSOC_FLAG_ROOTS_FOUND

root conflict is found

MESH_ASSOC_FLAG_ROOT_FIXED

fixed root

MESH_PS_DEVICE_DUTY_REQUEST

Mesh PS (Power Save) duty cycle type.

requests to join a network PS without specifying a device duty cycle. After the device joins the network, a network duty cycle will be provided by the network

MESH_PS_DEVICE_DUTY_DEMAND

requests to join a network PS and specifies a demanded device duty cycle

MESH_PS_NETWORK_DUTY_MASTER

indicates the device is the NWK-DUTY-MASTER (network duty cycle master)

MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE

Mesh PS (Power Save) duty cycle applied rule.

MESH_PS_NETWORK_DUTY_APPLIED_UPLINK

MESH_INIT_CONFIG_DEFAULT()

Type Definitions

typedef *mesh_addr_t* **mesh_event_root_address_t**

Root address.

typedef *wifi_event_sta_disconnected_t* **mesh_event_disconnected_t**

Parent disconnected information.

typedef *wifi_event_ap_staconnected_t* **mesh_event_child_connected_t**

Child connected information.

typedef *wifi_event_ap_stadisconnected_t* **mesh_event_child_disconnected_t**

Child disconnected information.

typedef *wifi_event_sta_connected_t* **mesh_event_router_switch_t**

New router information.

Enumerations

enum **mesh_event_id_t**

Enumerated list of mesh event id.

Values:

enumerator **MESH_EVENT_STARTED**

mesh is started

enumerator **MESH_EVENT_STOPPED**

mesh is stopped

enumerator **MESH_EVENT_CHANNEL_SWITCH**

channel switch

enumerator **MESH_EVENT_CHILD_CONNECTED**

a child is connected on softAP interface

enumerator **MESH_EVENT_CHILD_DISCONNECTED**

a child is disconnected on softAP interface

enumerator **MESH_EVENT_ROUTING_TABLE_ADD**

routing table is changed by adding newly joined children

enumerator **MESH_EVENT_ROUTING_TABLE_REMOVE**

routing table is changed by removing leave children

enumerator **MESH_EVENT_PARENT_CONNECTED**

parent is connected on station interface

enumerator **MESH_EVENT_PARENT_DISCONNECTED**

parent is disconnected on station interface

enumerator **MESH_EVENT_NO_PARENT_FOUND**

no parent found

enumerator **MESH_EVENT_LAYER_CHANGE**

layer changes over the mesh network

enumerator **MESH_EVENT_TODS_STATE**

state represents whether the root is able to access external IP network. This state is a manual event that needs to be triggered with esp_mesh_post_toDS_state().

enumerator **MESH_EVENT_VOTE_STARTED**

the process of voting a new root is started either by children or by the root

enumerator **MESH_EVENT_VOTE_STOPPED**

the process of voting a new root is stopped

enumerator **MESH_EVENT_ROOT_ADDRESS**

the root address is obtained. It is posted by mesh stack automatically.

enumerator **MESH_EVENT_ROOT_SWITCH_REQ**

root switch request sent from a new voted root candidate

enumerator **MESH_EVENT_ROOT_SWITCH_ACK**

root switch acknowledgment responds the above request sent from current root

enumerator **MESH_EVENT_ROOT_ASKED_YIELD**

the root is asked yield by a more powerful existing root. If self organized is disabled and this device is specified to be a root by users, users should set a new parent for this device. if self organized is enabled, this device will find a new parent by itself, users could ignore this event.

enumerator **MESH_EVENT_ROOT_FIXED**

when devices join a network, if the setting of Fixed Root for one device is different from that of its parent, the device will update the setting the same as its parent's. Fixed Root Setting of each device is variable as that setting changes of the root.

enumerator **MESH_EVENT_SCAN_DONE**

if self-organized networking is disabled, user can call esp_wifi_scan_start() to trigger this event, and add the corresponding scan done handler in this event.

enumerator **MESH_EVENT_NETWORK_STATE**

network state, such as whether current mesh network has a root.

enumerator **MESH_EVENT_STOP_RECONNECTION**

the root stops reconnecting to the router and non-root devices stop reconnecting to their parents.

enumerator **MESH_EVENT_FIND_NETWORK**

when the channel field in mesh configuration is set to zero, mesh stack will perform a full channel scan to find a mesh network that can join, and return the channel value after finding it.

enumerator **MESH_EVENT_ROUTER_SWITCH**

if users specify BSSID of the router in mesh configuration, when the root connects to another router with the same SSID, this event will be posted and the new router information is attached.

enumerator **MESH_EVENT_PS_PARENT_DUTY**

parent duty

enumerator **MESH_EVENT_PS_CHILD_DUTY**

child duty

enumerator **MESH_EVENT_PS_DEVICE_DUTY**

device duty

enumerator **MESH_EVENT_MAX**

enum **mesh_type_t**

Device type.

Values:

enumerator **MESH_IDLE**

hasn't joined the mesh network yet

enumerator **MESH_ROOT**

the only sink of the mesh network. Has the ability to access external IP network

enumerator **MESH_NODE**

intermediate device. Has the ability to forward packets over the mesh network

enumerator **MESH_LEAF**

has no forwarding ability

enumerator **MESH_STA**

connect to router with a standlone Wi-Fi station mode, no network expansion capability

enum **mesh_proto_t**

Protocol of transmitted application data.

Values:

enumerator **MESH_PROTO_BIN**

binary

enumerator **MESH_PROTO_HTTP**

HTTP protocol

enumerator **MESH_PROTO_JSON**

JSON format

enumerator **MESH_PROTO_MQTT**

MQTT protocol

enumerator **MESH_PROTO_AP**

IP network mesh communication of node' s AP interface

enumerator **MESH_PROTO_STA**

IP network mesh communication of node' s STA interface

enum **mesh_tos_t**

For reliable transmission, mesh stack provides three type of services.

Values:

enumerator **MESH_TOS_P2P**

provide P2P (point-to-point) retransmission on mesh stack by default

enumerator **MESH_TOS_E2E**

provide E2E (end-to-end) retransmission on mesh stack (Unimplemented)

enumerator **MESH_TOS_DEF**

no retransmission on mesh stack

enum **mesh_vote_reason_t**

Vote reason.

Values:

enumerator **MESH_VOTE_REASON_ROOT_INITIATED**

vote is initiated by the root

enumerator **MESH_VOTE_REASON_CHILD_INITIATED**

vote is initiated by children

enum **mesh_disconnect_reason_t**

Mesh disconnect reason code.

Values:

enumerator **MESH_REASON_CYCLIC**

cyclic is detected

enumerator **MESH_REASON_PARENT_IDLE**

parent is idle

enumerator **MESH_REASON_LEAF**

the connected device is changed to a leaf

enumerator **MESH_REASON_DIFF_ID**

in different mesh ID

enumerator **MESH_REASON_ROOTS**

root conflict is detected

enumerator **MESH_REASON_PARENT_STOPPED**

parent has stopped the mesh

enumerator **MESH_REASON_SCAN_FAIL**

scan fail

enumerator **MESH_REASON_IE_UNKNOWN**

unknown IE

enumerator **MESH_REASON_WAIVE_ROOT**

waive root

enumerator **MESH_REASON_PARENT_WORSE**

parent with very poor RSSI

enumerator **MESH_REASON_EMPTY_PASSWORD**

use an empty password to connect to an encrypted parent

enumerator **MESH_REASON_PARENT_UNENCRYPTED**

connect to an unencrypted parent/router

enum **esp_mesh_topology_t**

Mesh topology.

Values:

enumerator **MESH_TOPO_TREE**

tree topology

enumerator **MESH_TOPO_CHAIN**

chain topology

enum **mesh_event_toDS_state_t**

The reachability of the root to a DS (distribute system)

Values:

enumerator **MESH_TODS_UNREACHABLE**

the root isn't able to access external IP network

enumerator **MESH_TODS_REACHABLE**

the root is able to access external IP network

SmartConfig

The SmartConfig™ is a provisioning technology developed by TI to connect a new Wi-Fi device to a Wi-Fi network. It uses a mobile app to broadcast the network credentials from a smartphone, or a tablet, to an un-provisioned Wi-Fi device.

The advantage of this technology is that the device does not need to directly know SSID or password of an Access Point (AP). This information is provided using the smartphone. This is particularly important to headless device and systems, due to their lack of a user interface.

If you are looking for other options to provision your ESP32 devices, check *Provisioning API*.

Application Example Connect ESP32 to target AP using SmartConfig: wifi/smart_config.

API Reference

Header File

- components/esp_wifi/include/esp_smartconfig.h

Functions

const char ***esp_smartconfig_get_version**(void)

Get the version of SmartConfig.

Returns

- SmartConfig version const char.

esp_err_t **esp_smartconfig_start**(const *smartconfig_start_config_t* *config)

Start SmartConfig, config ESP device to connect AP. You need to broadcast information by phone APP. Device sniffer special packets from the air that containing SSID and password of target AP.

Attention 1. This API can be called in station or softAP-station mode.
Attention 2. Can not call esp_smartconfig_start twice before it finish, please call esp_smartconfig_stop first.

Parameters **config** –pointer to smartconfig start configure structure
Returns

- ESP_OK: succeed
- others: fail

esp_err_t **esp_smartconfig_stop**(void)

Stop SmartConfig, free the buffer taken by esp_smartconfig_start.

Attention Whether connect to AP succeed or not, this API should be called to free memory taken by smartconfig_start.

Returns
- ESP_OK: succeed
- others: fail

esp_err_t **esp_esptouch_set_timeout**(uint8_t time_s)

Set timeout of SmartConfig process.

Attention Timing starts from SC_STATUS_FIND_CHANNEL status. SmartConfig will restart if timeout.

Parameters **time_s** –range 15s~255s, offset:45s.
Returns
- ESP_OK: succeed
- others: fail

esp_err_t **esp_smartconfig_set_type**(*smartconfig_type_t* type)

Set protocol type of SmartConfig.

Attention If users need to set the SmartConfig type, please set it before calling esp_smartconfig_start.

Parameters **type** –Choose from the smartconfig_type_t.
Returns
- ESP_OK: succeed
- others: fail

esp_err_t **esp_smartconfig_fast_mode**(bool enable)

Set mode of SmartConfig. default normal mode.

Attention 1. Please call it before API esp_smartconfig_start.
Attention 2. Fast mode have corresponding APP(phone).
Attention 3. Two mode is compatible.

Parameters **enable** –false-disable(default); true-enable;
Returns
- ESP_OK: succeed
- others: fail

esp_err_t **esp_smartconfig_get_rvd_data**(uint8_t *rvd_data, uint8_t len)

Get reserved data of ESPTouch v2.

Parameters
- **rvd_data** –reserved data
- **len** –length of reserved data

Returns
- ESP_OK: succeed
- others: fail

Structures

struct **smartconfig_event_got_ssid_pswd_t**

Argument structure for SC_EVENT_GOT_SSID_PSWD event

Public Members

uint8_t **ssid**[32]

SSID of the AP. Null terminated string.

uint8_t **password**[64]

Password of the AP. Null terminated string.

bool **bssid_set**

whether set MAC address of target AP or not.

uint8_t **bssid**[6]

MAC address of target AP.

smartconfig_type_t **type**

Type of smartconfig(ESPTouch or AirKiss).

uint8_t **token**

Token from cellphone which is used to send ACK to cellphone.

uint8_t **cellphone_ip**[4]

IP address of cellphone.

struct **smartconfig_start_config_t**

Configure structure for esp_smartconfig_start

Public Members

bool **enable_log**

Enable smartconfig logs.

bool **esp_touch_v2_enable_crypt**

Enable ESPTouch v2 crypt.

char ***esp_touch_v2_key**

ESPTouch v2 crypt key, len should be 16.

Macros

SMARTCONFIG_START_CONFIG_DEFAULT()

Enumerations

enum **smartconfig_type_t**

Values:

enumerator **SC_TYPE_ESPTOUCH**

protocol: ESPTouch

enumerator **SC_TYPE_AIRKISS**

protocol: AirKiss

enumerator **SC_TYPE_ESPTOUCH_AIRKISS**

protocol: ESPTouch and AirKiss

enumerator **SC_TYPE_ESPTOUCH_V2**

protocol: ESPTouch v2

enum **smartconfig_event_t**

Smartconfig event declarations

Values:

enumerator **SC_EVENT_SCAN_DONE**

ESP32 station smartconfig has finished to scan for APs

enumerator **SC_EVENT_FOUND_CHANNEL**

ESP32 station smartconfig has found the channel of the target AP

enumerator **SC_EVENT_GOT_SSID_PSWD**

ESP32 station smartconfig got the SSID and password

enumerator **SC_EVENT_SEND_ACK_DONE**

ESP32 station smartconfig has sent ACK to cellphone

Wi-Fi

Introduction The Wi-Fi libraries provide support for configuring and monitoring the ESP32 Wi-Fi networking functionality. This includes configuration for:

- Station mode (aka STA mode or Wi-Fi client mode). ESP32 connects to an access point.
- AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP32.
- Station/AP-coexistence mode (ESP32 is concurrently an access point and a station connected to another access point).
- Various security modes for the above (WPA, WPA2, WPA3, etc.)
- Scanning for access points (active & passive scanning).
- Promiscuous mode for monitoring of IEEE802.11 Wi-Fi packets.

Application Examples The wifi directory of ESP-IDF examples contains the following applications:

- Code examples for Wi-Fi.
- A simple esp-idf-template application to demonstrate a minimal IDF project structure.

API Reference

Header File

- components/esp_wifi/include/esp_wifi.h

Functions

esp_err_t **esp_wifi_init** (const *wifi_init_config_t* *config)

Initialize WiFi Allocate resource for WiFi driver, such as WiFi control structure, RX/TX buffer, WiFi NVS structure etc. This WiFi also starts WiFi task.

Attention 1. This API must be called before all other WiFi API can be called

Attention 2. Always use WIFI_INIT_CONFIG_DEFAULT macro to initialize the configuration to default values, this can guarantee all the fields get correct value when more fields are added into *wifi_init_config_t* in future release. If you want to set your own initial values, overwrite the default values which are set by WIFI_INIT_CONFIG_DEFAULT. Please be notified that the field 'magic' of *wifi_init_config_t* should always be WIFI_INIT_CONFIG_MAGIC!

Parameters **config** –pointer to WiFi initialized configuration structure; can point to a temporary variable.

Returns

- ESP_OK: succeed
- ESP_ERR_NO_MEM: out of memory
- others: refer to error code esp_err.h

esp_err_t **esp_wifi_deinit** (void)

Deinit WiFi Free all resource allocated in esp_wifi_init and stop WiFi task.

Attention 1. This API should be called if you want to remove WiFi driver from the system

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_mode** (*wifi_mode_t* mode)

Set the WiFi operating mode.

```
Set the WiFi operating mode as station, soft-AP or station+soft-AP,
The default mode is station mode.
```

Parameters **mode** –WiFi operating mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error code in esp_err.h

esp_err_t **esp_wifi_get_mode** (*wifi_mode_t* *mode)

Get current operating mode of WiFi.

Parameters **mode** –**[out]** store current WiFi mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_start**(void)

Start WiFi according to current configuration If mode is WIFI_MODE_STA, it create station control block and start station If mode is WIFI_MODE_AP, it create soft-AP control block and start soft-AP If mode is WIFI_MODE_APSTA, it create soft-AP and station control block and start soft-AP and station.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NO_MEM: out of memory
- ESP_ERR_WIFI_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_stop**(void)

Stop WiFi If mode is WIFI_MODE_STA, it stop station and free station control block If mode is WIFI_MODE_AP, it stop soft-AP and free soft-AP control block If mode is WIFI_MODE_APSTA, it stop station/soft-AP and free station/soft-AP control block.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_restore**(void)

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- esp_wifi_set_bandwidth,
- esp_wifi_set_protocol,
- esp_wifi_set_config related
- esp_wifi_set_mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_connect**(void)

Connect the ESP32 WiFi station to the AP.

Attention 1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode

Attention 2. If the ESP32 is connected to an AP, call esp_wifi_disconnect to disconnect.

Attention 3. The scanning triggered by esp_wifi_scan_start() will not be effective until connection between ESP32 and the AP is established. If ESP32 is scanning and connecting at the same time, ESP32 will abort scanning and return a warning message and error number ESP_ERR_WIFI_STATE.

Attention 4. This API attempts to connect to an Access Point (AP) only once. To enable reconnection in case of a connection failure, please use the 'failure_retry_cnt' feature in the '*wifi_sta_config_t*'. Users are suggested to implement reconnection logic in their application for scenarios where the specified AP does not exist, or reconnection is desired after the device has received a disconnect event.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP_ERR_WIFI_SSID: SSID of AP which station connects is invalid

esp_err_t **esp_wifi_disconnect** (void)

Disconnect the ESP32 WiFi station from the AP.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi was not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_clear_fast_connect** (void)

Currently this API is just an stub API.

Returns

- ESP_OK: succeed
- others: fail

esp_err_t **esp_wifi_deauth_sta** (uint16_t aid)

deauthenticate all stations or associated id equals to aid

Parameters **aid** –when aid is 0, deauthenticate all stations, otherwise deauthenticate station whose associated id is aid

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_MODE: WiFi mode is wrong

esp_err_t **esp_wifi_scan_start** (const *wifi_scan_config_t* *config, bool block)

Scan all available APs.

Attention If this API is called, the found APs are stored in WiFi driver dynamic allocated memory. And then can be freed in esp_wifi_scan_get_ap_records(), esp_wifi_scan_get_ap_record() or esp_wifi_clear_ap_list(), so call any one to free the memory once the scan is done.

Attention The values of maximum active scan time and passive scan time per channel are limited to 1500 milliseconds. Values above 1500ms may cause station to disconnect from AP and are not recommended.

Parameters

- **config** –configuration settings for scanning, if set to NULL default settings will be used of which default values are show_hidden:false, scan_type:active, scan_time.active.min:0, scan_time.active.max:120 miliseconds, scan_time.passive:360 miliseconds
- **block** –if block is true, this API will block the caller until the scan is done, otherwise it will return immediately

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_ERR_WIFI_TIMEOUT: blocking scan is timeout
- ESP_ERR_WIFI_STATE: wifi still connecting when invoke esp_wifi_scan_start
- others: refer to error code in esp_err.h

esp_err_t **esp_wifi_scan_stop** (void)

Stop the scan in process.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

esp_err_t **esp_wifi_scan_get_ap_num**(uint16_t *number)

Get number of APs found in last scan.

Attention This API can only be called when the scan is completed, otherwise it may get wrong value.

Parameters **number** –**[out]** store number of APs found in last scan

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_scan_get_ap_records**(uint16_t *number, *wifi_ap_record_t* *ap_records)

Get AP list found in last scan.

Attention This API will free all memory occupied by scanned AP list.

Parameters

- **number** –**[inout]** As input param, it stores max AP number ap_records can hold. As output param, it receives the actual AP number this API returns.
- **ap_records** –*wifi_ap_record_t* array to hold the found APs

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NO_MEM: out of memory

esp_err_t **esp_wifi_scan_get_ap_record**(*wifi_ap_record_t* *ap_record)

Get one AP record from the scanned AP list.

Attention Different from esp_wifi_scan_get_ap_records(), this API only gets one AP record from the scanned AP list each time. This API will free the memory of one AP record, if the user doesn't get all records in the scannned AP list, then needs to call esp_wifi_clear_ap_list() to free the remaining memory.

Parameters **ap_record** –**[out]** pointer to one AP record

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_FAIL: scan APs is NULL, means all AP records fetched or no AP found

esp_err_t **esp_wifi_clear_ap_list**(void)

Clear AP list found in last scan.

Attention This API will free all memory occupied by scanned AP list. When the obtained AP list fails, AP records must be cleared,otherwise it may cause memory leakage.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_sta_get_ap_info**(*wifi_ap_record_t* *ap_info)

Get information of AP which the ESP32 station is associated with.

Attention When the obtained country information is empty, it means that the AP does not carry country information

Parameters **ap_info** –the *wifi_ap_record_t* to hold AP information sta can get the connected ap's phy mode info through the struct member phy_11b, phy_11g, phy_11n, phy_lr in the *wifi_ap_record_t* struct. For example, phy_11b = 1 imply that ap support 802.11b mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_CONN: The station interface don't initialized
- ESP_ERR_WIFI_NOT_CONNECT: The station is in disconnect status

esp_err_t **esp_wifi_set_ps**(*wifi_ps_type_t* type)

Set current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Parameters **type** –power save type

Returns ESP_OK: succeed

esp_err_t **esp_wifi_get_ps**(*wifi_ps_type_t* *type)

Get current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Parameters **type** –**[out]** store current power save type

Returns ESP_OK: succeed

esp_err_t **esp_wifi_set_protocol**(*wifi_interface_t* ifx, uint8_t protocol_bitmap)

Set protocol type of specified interface The default protocol is (WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROT

Attention Support 802.11b or 802.11bg or 802.11bgn or LR mode

Parameters

- **ifx** –interfaces
- **protocol_bitmap** –WiFi protocol bitmap

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- others: refer to error codes in esp_err.h

esp_err_t **esp_wifi_get_protocol**(*wifi_interface_t* ifx, uint8_t *protocol_bitmap)

Get the current protocol bitmap of the specified interface.

Parameters

- **ifx** –interface
- **protocol_bitmap** –**[out]** store current WiFi protocol bitmap of interface ifx

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error codes in esp_err.h

esp_err_t **esp_wifi_set_bandwidth**(*wifi_interface_t* ifx, *wifi_bandwidth_t* bw)

Set the bandwidth of ESP32 specified interface.

Attention 1. API return false if try to configure an interface that is not enabled

Attention 2. WIFI_BW_HT40 is supported only when the interface support 11N

Parameters

- **ifx** –interface to be configured
- **bw** –bandwidth

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error codes in esp_err.h

esp_err_t **esp_wifi_get_bandwidth**(*wifi_interface_t* ifx, *wifi_bandwidth_t* *bw)

Get the bandwidth of ESP32 specified interface.

Attention 1. API return false if try to get a interface that is not enable

Parameters

- **ifx** –interface to be configured
- **bw** –**[out]** store bandwidth of interface ifx

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_channel**(uint8_t primary, *wifi_second_chan_t* second)

Set primary/secondary channel of ESP32.

Attention 1. This API should be called after esp_wifi_start() and before esp_wifi_stop()

Attention 2. When ESP32 is in STA mode, this API should not be called when STA is scanning or connecting to an external AP

Attention 3. When ESP32 is in softAP mode, this API should not be called when softAP has connected to external STAs

Attention 4. When ESP32 is in STA+softAP mode, this API should not be called when in the scenarios described above

Attention 5. The channel info set by this API will not be stored in NVS. So If you want to remeber the channel used before wifi stop, you need to call this API again after wifi start, or you can call `esp_wifi_set_config()` to store the channel info in NVS.

Parameters

- **primary** –for HT20, primary is the channel number, for HT40, primary is the primary channel
- **second** –for HT20, second is ignored, for HT40, second is the second channel

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

esp_err_t **esp_wifi_get_channel**(uint8_t *primary, *wifi_second_chan_t* *second)

Get the primary/secondary channel of ESP32.

Attention 1. API return false if try to get a interface that is not enable

Parameters

- **primary** –store current primary channel
- **second** –**[out]** store current second channel

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_country**(const *wifi_country_t* *country)

configure country info

Attention 1. It is discouraged to call this API since this doesn't validate the per-country rules, it's up to the user to fill in all fields according to local regulations. Please use esp_wifi_set_country_code instead.

Attention 2. The default country is "01" (world safe mode) {.cc="01", .schan=1, .nchan=11, .policy=WIFI_COUNTRY_POLICY_AUTO}.

Attention 3. The third octet of country code string is one of the following: ' ', 'O', 'I', 'X', otherwise it is considered as ' '.

Attention 4. When the country policy is WIFI_COUNTRY_POLICY_AUTO, the country info of the AP to which the station is connected is used. E.g. if the configured country info is {.cc="US", .schan=1, .nchan=11} and the country info of the AP to which the station is connected is {.cc="JP", .schan=1, .nchan=14} then the country info that will be used is {.cc="JP", .schan=1, .nchan=14}. If the station disconnected from the AP the country info is set back to the country info of the station automatically, {.cc="US", .schan=1, .nchan=11} in the example.

Attention 5. When the country policy is WIFI_COUNTRY_POLICY_MANUAL, then the configured country info is used always.

Attention 6. When the country info is changed because of configuration or because the station connects to a different external AP, the country IE in probe response/beacon of the soft-AP is also changed.

Attention 7. The country configuration is stored into flash.

Attention 8. When this API is called, the PHY init data will switch to the PHY init data type corresponding to the country info.

Parameters **country** –the configured country info

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_get_country**(*wifi_country_t* *country)

get the current country info

Parameters **country** –country info

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_mac**(*wifi_interface_t* ifx, const uint8_t mac[6])

Set MAC address of the ESP32 WiFi station or the soft-AP interface.

Attention 1. This API can only be called when the interface is disabled

Attention 2. ESP32 soft-AP and station have different MAC addresses, do not set them to be the same.

Attention 3. The bit 0 of the first byte of ESP32 MAC address can not be 1. For example, the MAC address can set to be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX:XX".

Parameters

- **ifx** –interface
- **mac** –the MAC address

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MAC: invalid mac address
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- others: refer to error codes in esp_err.h

esp_err_t **esp_wifi_get_mac**(*wifi_interface_t* ifx, uint8_t mac[6])

Get mac of specified interface.

Parameters

- **ifx** –interface
- **mac** –**[out]** store mac of the interface ifx

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface

esp_err_t **esp_wifi_set_promiscuous_rx_cb**(*wifi_promiscuous_cb_t* cb)

Register the RX callback function in the promiscuous mode.

Each time a packet is received, the registered callback function will be called.

Parameters **cb** –callback

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_promiscuous**(bool en)

Enable the promiscuous mode.

Parameters **en** –false - disable, true - enable

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_get_promiscuous**(bool *en)

Get the promiscuous mode.

Parameters **en** –**[out]** store the current status of promiscuous mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_promiscuous_filter**(const *wifi_promiscuous_filter_t* *filter)

Enable the promiscuous mode packet type filter.

Note: The default filter is to filter all packets except WIFI_PKT_MISC

Parameters **filter** –the packet type filtered in promiscuous mode.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_get_promiscuous_filter**(*wifi_promiscuous_filter_t* *filter)

Get the promiscuous filter.

Parameters **filter** –**[out]** store the current status of promiscuous filter

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_promiscuous_ctrl_filter**(const *wifi_promiscuous_filter_t* *filter)

Enable subtype filter of the control packet in promiscuous mode.

Note: The default filter is to filter none control packet.

Parameters **filter** –the subtype of the control packet filtered in promiscuous mode.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_get_promiscuous_ctrl_filter**(*wifi_promiscuous_filter_t* *filter)

Get the subtype filter of the control packet in promiscuous mode.

Parameters **filter** –**[out]** store the current status of subtype filter of the control packet in promiscuous mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_config**(*wifi_interface_t* interface, *wifi_config_t* *conf)

Set the configuration of the ESP32 STA or AP.

Attention 1. This API can be called only when specified interface is enabled, otherwise, API fail

Attention 2. For station configuration, bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

Attention 3. ESP32 is limited to only one channel, so when in the soft-AP+station mode, the soft-AP will adjust its channel automatically to be the same as the channel of the ESP32 station.

Attention 4. The configuration will be stored in NVS

Parameters

- **interface** –interface
- **conf** –station or soft-AP configuration

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MODE: invalid mode
- ESP_ERR_WIFI_PASSWORD: invalid password
- ESP_ERR_WIFI_NVS: WiFi internal NVS error
- others: refer to the error code in esp_err.h

esp_err_t **esp_wifi_get_config**(*wifi_interface_t* interface, *wifi_config_t* *conf)

Get configuration of specified interface.

Parameters

- **interface** –interface
- **conf** –**[out]** station or soft-AP configuration

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface

esp_err_t **esp_wifi_ap_get_sta_list**(*wifi_sta_list_t* *sta)

Get STAs associated with soft-AP.

Attention SSC only API

Parameters **sta** –**[out]** station list ap can get the connected sta's phy mode info through the struct member phy_11b, phy_11g, phy_11n, phy_lr in the *wifi_sta_info_t* struct. For example, phy_11b = 1 imply that sta support 802.11b mode

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_WIFI_CONN: WiFi internal error, the station/soft-AP control block is invalid

esp_err_t **esp_wifi_ap_get_sta_aid**(const uint8_t mac[6], uint16_t *aid)

Get AID of STA connected with soft-AP.

Parameters

- **mac** –STA's mac address
- **aid** –**[out]** Store the AID corresponding to STA mac

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NOT_FOUND: Requested resource not found
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_WIFI_CONN: WiFi internal error, the station/soft-AP control block is invalid

esp_err_t **esp_wifi_set_storage**(*wifi_storage_t* storage)

Set the WiFi API configuration storage type.

Attention 1. The default value is WIFI_STORAGE_FLASH

Parameters **storage** –: storage type

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_vendor_ie**(bool enable, *wifi_vendor_ie_type_t* type, *wifi_vendor_ie_id_t* idx, const void *vnd_ie)

Set 802.11 Vendor-Specific Information Element.

Parameters
- **enable** –If true, specified IE is enabled. If false, specified IE is removed.
- **type** –Information Element type. Determines the frame type to associate with the IE.
- **idx** –Index to set or clear. Each IE type can be associated with up to two elements (indices 0 & 1).
- **vnd_ie** –Pointer to vendor specific element data. First 6 bytes should be a header with fields matching *vendor_ie_data_t*. If enable is false, this argument is ignored and can be NULL. Data does not need to remain valid after the function returns.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init()
- ESP_ERR_INVALID_ARG: Invalid argument, including if first byte of vnd_ie is not WIFI_VENDOR_IE_ELEMENT_ID (0xDD) or second byte is an invalid length.
- ESP_ERR_NO_MEM: Out of memory

esp_err_t **esp_wifi_set_vendor_ie_cb**(*esp_vendor_ie_cb_t* cb, void *ctx)

Register Vendor-Specific Information Element monitoring callback.

Parameters
- **cb** –Callback function
- **ctx** –Context argument, passed to callback function.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_max_tx_power**(int8_t power)

Set maximum transmitting power after WiFi start.

Attention 1. Maximum power before wifi startup is limited by PHY init data bin.

Attention 2. The value set by this API will be mapped to the max_tx_power of the structure *wifi_country_t* variable.

Attention 3. Mapping Table {Power, max_tx_power} = {{8, 2}, {20, 5}, {28, 7}, {34, 8}, {44, 11}, {52, 13}, {56, 14}, {60, 15}, {66, 16}, {72, 18}, {80, 20}}.

Attention 4. Param power unit is 0.25dBm, range is [8, 84] corresponding to 2dBm - 20dBm.

Attention 5. Relationship between set value and actual value. As follows: {set value range, actual value} = {{[8, 19],8}, {[20, 27],20}, {[28, 33],28}, {[34, 43],34}, {[44, 51],44}, {[52, 55],52}, {[56, 59],56}, {[60, 65],60}, {[66, 71],66}, {[72, 79],72}, {[80, 84],80}}.

Parameters **power** –Maximum WiFi transmitting power.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is out of range

esp_err_t **esp_wifi_get_max_tx_power**(int8_t *power)

Get maximum transmiting power after WiFi start.

Parameters **power** –Maximum WiFi transmitting power, unit is 0.25dBm.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_set_event_mask**(uint32_t mask)

Set mask to enable or disable some WiFi events.

Attention 1. Mask can be created by logical OR of various WIFI_EVENT_MASK_ constants. Events which have corresponding bit set in the mask will not be delivered to the system event handler.

Attention 2. Default WiFi event mask is WIFI_EVENT_MASK_AP_PROBEREQRECVED.

Attention 3. There may be lots of stations sending probe request data around. Don't unmask this event unless you need to receive probe request data.

Parameters **mask** –WiFi event mask.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_get_event_mask**(uint32_t *mask)

Get mask of WiFi events.

Parameters **mask** –WiFi event mask.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_80211_tx**(*wifi_interface_t* ifx, const void *buffer, int len, bool en_sys_seq)

Send raw ieee80211 data.

Attention Currently only support for sending beacon/probe request/probe response/action and non-QoS data frame

Parameters

- **ifx** –interface if the Wi-Fi mode is Station, the ifx should be WIFI_IF_STA. If the Wi-Fi mode is SoftAP, the ifx should be WIFI_IF_AP. If the Wi-Fi mode is Station+SoftAP, the ifx should be WIFI_IF_STA or WIFI_IF_AP. If the ifx is wrong, the API returns ESP_ERR_WIFI_IF.
- **buffer** –raw ieee80211 buffer
- **len** –the length of raw buffer, the len must be <= 1500 Bytes and >= 24 Bytes
- **en_sys_seq** –indicate whether use the internal sequence number. If en_sys_seq is false, the sequence in raw buffer is unchanged, otherwise it will be overwritten by WiFi driver with the system sequence number. Generally, if esp_wifi_80211_tx is called before the Wi-Fi connection has been set up, both en_sys_seq==true and en_sys_seq==false are fine. However, if the API is called after the Wi-Fi connection has been set up, en_sys_seq must be true, otherwise ESP_ERR_INVALID_ARG is returned.

Returns

- ESP_OK: success
- ESP_ERR_WIFI_IF: Invalid interface
- ESP_ERR_INVALID_ARG: Invalid parameter
- ESP_ERR_WIFI_NO_MEM: out of memory

esp_err_t **esp_wifi_set_csi_rx_cb**(*wifi_csi_cb_t* cb, void *ctx)

Register the RX callback function of CSI data.

```
Each time a CSI data is received, the callback function will be called.
```

Parameters
- **cb** –callback
- **ctx** –context argument, passed to callback function

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_csi_config**(const *wifi_csi_config_t* *config)

Set CSI data configuration.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters **config** –configuration

esp_err_t **esp_wifi_set_csi**(bool en)

Enable or disable CSI.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters **en** –true - enable, false - disable

esp_err_t **esp_wifi_set_ant_gpio**(const *wifi_ant_gpio_config_t* *config)

Set antenna GPIO configuration.

Parameters **config** –Antenna GPIO configuration.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: Invalid argument, e.g. parameter is NULL, invalid GPIO number etc

esp_err_t **esp_wifi_get_ant_gpio**(*wifi_ant_gpio_config_t* *config)

Get current antenna GPIO configuration.

Parameters **config** –Antenna GPIO configuration.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL

esp_err_t **esp_wifi_set_ant**(const *wifi_ant_config_t* *config)

Set antenna configuration.

Parameters **config** –Antenna configuration.

Returns
- ESP_OK: succeed

- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: Invalid argument, e.g. parameter is NULL, invalid antenna mode or invalid GPIO number

esp_err_t **esp_wifi_get_ant**(*wifi_ant_config_t* *config)

Get current antenna configuration.

Parameters **config** –Antenna configuration.
Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL

int64_t **esp_wifi_get_tsf_time**(*wifi_interface_t* interface)

Get the TSF time In Station mode or SoftAP+Station mode if station is not connected or station doesn't receive at least one beacon after connected, will return 0.

Attention Enabling power save may cause the return value inaccurate, except WiFi modem sleep

Parameters **interface** –The interface whose tsf_time is to be retrieved.
Returns 0 or the TSF time

esp_err_t **esp_wifi_set_inactive_time**(*wifi_interface_t* ifx, uint16_t sec)

Set the inactive time of the ESP32 STA or AP.

Attention 1. For Station, If the station does not receive a beacon frame from the connected SoftAP during the inactive time, disconnect from SoftAP. Default 6s.
Attention 2. For SoftAP, If the softAP doesn't receive any data from the connected STA during inactive time, the softAP will force deauth the STA. Default is 300s.
Attention 3. The inactive time configuration is not stored into flash

Parameters
- **ifx** –interface to be configured.
- **sec** –Inactive time. Unit seconds.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument, For Station, if sec is less than 3. For SoftAP, if sec is less than 10.

esp_err_t **esp_wifi_get_inactive_time**(*wifi_interface_t* ifx, uint16_t *sec)

Get inactive time of specified interface.

Parameters
- **ifx** –Interface to be configured.
- **sec** –Inactive time. Unit seconds.

Returns
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_statis_dump**(uint32_t modules)

Dump WiFi statistics.

Parameters **modules** –statistic modules to be dumped
Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_set_rssi_threshold**(int32_t rssi)

Set RSSI threshold, if average rssi gets lower than threshold, WiFi task will post event WIFI_EVENT_STA_BSS_RSSI_LOW.

Attention If the user wants to receive another WIFI_EVENT_STA_BSS_RSSI_LOW event after receiving one, this API needs to be called again with an updated/same RSSI threshold.

Parameters **rssi** –threshold value in dbm between -100 to 10 Note that in some rare cases where signal strength is very strong, rssi values can be slightly positive.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_ftm_initiate_session**(*wifi_ftm_initiator_cfg_t* *cfg)

Start an FTM Initiator session by sending FTM request If successful, event WIFI_EVENT_FTM_REPORT is generated with the result of the FTM procedure.

Attention 1. Use this API only in Station mode.

Attention 2. If FTM is initiated on a different channel than Station is connected in or internal SoftAP is started in, FTM defaults to a single burst in ASAP mode.

Parameters **cfg** –FTM Initiator session configuration

Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_ftm_end_session**(void)

End the ongoing FTM Initiator session.

Attention This API works only on FTM Initiator

Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_ftm_resp_set_offset**(int16_t offset_cm)

Set offset in cm for FTM Responder. An equivalent offset is calculated in picoseconds and added in TOD of FTM Measurement frame (T1).

Attention Use this API only in AP mode before performing FTM as responder

Parameters **offset_cm** –T1 Offset to be added in centimeters

Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_ftm_get_report**(*wifi_ftm_report_entry_t* *report, uint8_t num_entries)

Get FTM measurements report copied into a user provided buffer.

Attention 1. To get the FTM report, user first needs to allocate a buffer of size (sizeof(wifi_ftm_report_entry_t) * num_entries) where the API will fill up to num_entries valid FTM measurements in the buffer. Total number of entries can be found in the event WIFI_EVENT_FTM_REPORT as ftm_report_num_entries

Attention 2. The internal FTM report is freed upon use of this API which means the API can only be used once afer every FTM session initiated

Attention 3. Passing the buffer as NULL merely frees the FTM report

Parameters

- **report** –Pointer to the buffer for receiving the FTM report
- **num_entries** –Number of FTM report entries to be filled in the report

Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_config_11b_rate**(*wifi_interface_t* ifx, bool disable)

Enable or disable 11b rate of specified interface.

Attention 1. This API should be called after esp_wifi_init() and before esp_wifi_start().

Attention 2. Only when really need to disable 11b rate call this API otherwise don't call this.

Parameters

- **ifx** –Interface to be configured.
- **disable** –true means disable 11b rate while false means enable 11b rate.

Returns

- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_connectionless_module_set_wake_interval**(uint16_t wake_interval)

Set wake interval for connectionless modules to wake up periodically.

Attention 1. Only one wake interval for all connectionless modules.

Attention 2. This configuration could work at connected status. When ESP_WIFI_STA_DISCONNECTED_PM_ENABLE is enabled, this configuration could work at disconnected status.

Attention 3. Event WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START would be posted each time wake interval starts.

Attention 4. Recommend to configure interval in multiples of hundred. (e.g. 100ms)

Attention 5. Recommend to configure interval to ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE to get stable performance at coexistence mode.

Parameters **wake_interval** –Milliseconds after would the chip wake up, from 1 to 65535.

esp_err_t **esp_wifi_force_wakeup_acquire**(void)

Request extra reference of Wi-Fi radio. Wi-Fi keep active state(RF opened) to be able to receive packets.

Attention Please pair the use of `esp_wifi_force_wakeup_acquire` with `esp_wifi_force_wakeup_release`.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

esp_err_t **esp_wifi_force_wakeup_release**(void)

Release extra reference of Wi-Fi radio. Wi-Fi go to sleep state(RF closed) if no more use of radio.

Attention Please pair the use of `esp_wifi_force_wakeup_acquire` with `esp_wifi_force_wakeup_release`.

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

esp_err_t **esp_wifi_set_country_code**(const char *country, bool ieee80211d_enabled)

configure country

Attention 1. When ieee80211d_enabled, the country info of the AP to which the station is connected is used. E.g. if the configured country is US and the country info of the AP to which the station is connected is JP then the country info that will be used is JP. If the station disconnected from the AP the country info is set back to the country info of the station automatically, US in the example.

Attention 2. When ieee80211d_enabled is disabled, then the configured country info is used always.

Attention 3. When the country info is changed because of configuration or because the station connects to a different external AP, the country IE in probe response/beacon of the soft-AP is also changed.

Attention 4. The country configuration is stored into flash.

Attention 5. When this API is called, the PHY init data will switch to the PHY init data type corresponding to the country info.

Attention 6. Supported country codes are "01" (world safe mode) "AT" ," AU" ," BE" ," BG" ," BR" , "CA" ," CH" ," CN" ," CY" ," CZ" ," DE" ," DK" ," EE" ," ES" ," FI" ," FR" ," GB" ," GR" ," HK" ," HR" ," HU" , "IE" ," IN" ," IS" ," IT" ," JP" ," KR" ," LI" ," LT" ," LU" ," LV" ," MT" ," MX" ," NL" ," NO" ," NZ" ," PL" ," PT" , "RO" ," SE" ," SI" ," SK" ," TW" ," US"

Attention 7. When country code "01" (world safe mode) is set, SoftAP mode won't contain country IE.

Attention 8. The default country is "01" (world safe mode) and ieee80211d_enabled is TRUE.

Attention 9. The third octet of country code string is one of the following: ' ', 'O', 'I', 'X', otherwise it is considered as ' '.

Parameters

- **country** –the configured country ISO code
- **ieee80211d_enabled** –802.11d is enabled or not

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_get_country_code**(char *country)

get the current country code

Parameters **country** –country code

Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

esp_err_t **esp_wifi_config_80211_tx_rate**(*wifi_interface_t* ifx, *wifi_phy_rate_t* rate)

Config 80211 tx rate of specified interface.

Attention 1. This API should be called after esp_wifi_init() and before esp_wifi_start().

Parameters
- **ifx** –Interface to be configured.
- **rate** –Phy rate to be configured.

Returns
- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_disable_pmf_config**(*wifi_interface_t* ifx)

Disable PMF configuration for specified interface.

Attention This API should be called after esp_wifi_set_config() and before esp_wifi_start().

Parameters **ifx** –Interface to be configured.

Returns
- ESP_OK: succeed
- others: failed

esp_err_t **esp_wifi_sta_get_aid**(uint16_t *aid)

Get the Association id assigned to STA by AP.

Attention aid = 0 if station is not connected to AP.

Parameters **aid** –**[out]** store the aid

Returns
- ESP_OK: succeed

esp_err_t **esp_wifi_sta_get_negotiated_phymode**(*wifi_phy_mode_t* *phymode)

Get the negotiated phymode after connection.

Parameters **phymode** –**[out]** store the negotiated phymode.

Returns
- ESP_OK: succeed

esp_err_t **esp_wifi_sta_get_rssi**(int *rssi)

Get the rssi info after station connected to AP.

Attention This API should be called after station connected to AP.

Parameters **rssi** –store the rssi info received from last beacon.

Returns
- ESP_OK: succeed
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_FAIL: failed

Structures

struct **wifi_init_config_t**

WiFi stack configuration parameters passed to esp_wifi_init call.

Public Members

wifi_osi_funcs_t ***osi_funcs**

WiFi OS functions

wpa_crypto_funcs_t **wpa_crypto_funcs**

WiFi station crypto functions when connect

int **static_rx_buf_num**

WiFi static RX buffer number

int **dynamic_rx_buf_num**

WiFi dynamic RX buffer number

int **tx_buf_type**

WiFi TX buffer type

int **static_tx_buf_num**

WiFi static TX buffer number

int **dynamic_tx_buf_num**

WiFi dynamic TX buffer number

int **rx_mgmt_buf_type**

WiFi RX MGMT buffer type

int **rx_mgmt_buf_num**

WiFi RX MGMT buffer number

int **cache_tx_buf_num**

WiFi TX cache buffer number

int **csi_enable**

WiFi channel state information enable flag

int **ampdu_rx_enable**

WiFi AMPDU RX feature enable flag

int **ampdu_tx_enable**

WiFi AMPDU TX feature enable flag

int **amsdu_tx_enable**

WiFi AMSDU TX feature enable flag

int **nvs_enable**

WiFi NVS flash enable flag

int **nano_enable**

Nano option for printf/scan family enable flag

int **rx_ba_win**

WiFi Block Ack RX window size

int **wifi_task_core_id**

WiFi Task Core ID

int **beacon_max_len**

WiFi softAP maximum length of the beacon

int **mgmt_sbuf_num**

WiFi management short buffer number, the minimum value is 6, the maximum value is 32

uint64_t **feature_caps**

Enables additional WiFi features and capabilities

bool **sta_disconnected_pm**

WiFi Power Management for station at disconnected status

int **espnow_max_encrypt_num**

Maximum encrypt number of peers supported by espnow

int **magic**

WiFi init magic number, it should be the last field

Macros

ESP_ERR_WIFI_NOT_INIT

WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED

WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED

WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF

WiFi interface error

ESP_ERR_WIFI_MODE

WiFi mode error

ESP_ERR_WIFI_STATE

WiFi internal state error

ESP_ERR_WIFI_CONN

WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS

WiFi internal NVS module error

ESP_ERR_WIFI_MAC
MAC address is invalid

ESP_ERR_WIFI_SSID
SSID is invalid

ESP_ERR_WIFI_PASSWORD
Password is invalid

ESP_ERR_WIFI_TIMEOUT
Timeout error

ESP_ERR_WIFI_WAKE_FAIL
WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK
The caller would block

ESP_ERR_WIFI_NOT_CONNECT
Station still in disconnect status

ESP_ERR_WIFI_POST
Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE
Invalid WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE
Returned when WiFi is stopping

ESP_ERR_WIFI_NOT_ASSOC
The WiFi connection is not associated

ESP_ERR_WIFI_TX_DISALLOW
The WiFi TX is disallowed

ESP_ERR_WIFI_DISCARD
Discard frame

ESP_ERR_WIFI_ROC_IN_PROGRESS
ROC op is in progress

WIFI_STATIC_TX_BUFFER_NUM

WIFI_CACHE_TX_BUFFER_NUM

WIFI_DYNAMIC_TX_BUFFER_NUM

WIFI_RX_MGMT_BUF_NUM_DEF

WIFI_CSI_ENABLED

WIFI_AMPDU_RX_ENABLED

WIFI_AMPDU_TX_ENABLED

WIFI_AMSDU_TX_ENABLED

WIFI_NVS_ENABLED

WIFI_NANO_FORMAT_ENABLED

WIFI_INIT_CONFIG_MAGIC

WIFI_DEFAULT_RX_BA_WIN

WIFI_TASK_CORE_ID

WIFI_SOFTAP_BEACON_MAX_LEN

WIFI_MGMT_SBUF_NUM

WIFI_STA_DISCONNECTED_PM_ENABLED

WIFI_ENABLE_WPA3_SAE

WIFI_ENABLE_SPIRAM

WIFI_FTM_INITIATOR

WIFI_FTM_RESPONDER

CONFIG_FEATURE_WPA3_SAE_BIT

CONFIG_FEATURE_CACHE_TX_BUF_BIT

CONFIG_FEATURE_FTM_INITIATOR_BIT

CONFIG_FEATURE_FTM_RESPONDER_BIT

WIFI_FEATURE_CAPS

WIFI_INIT_CONFIG_DEFAULT()

ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE

Type Definitions

typedef void (*`wifi_promiscuous_cb_t`)(void *buf, *wifi_promiscuous_pkt_type_t* type)

The RX callback function in the promiscuous mode. Each time a packet is received, the callback function will be called.

Param buf Data received. Type of data in buffer (*wifi_promiscuous_pkt_t* or *wifi_pkt_rx_ctrl_t*) indicated by 'type' parameter.
Param type promiscuous packet type.

typedef void (*`esp_vendor_ie_cb_t`)(void *ctx, *wifi_vendor_ie_type_t* type, const uint8_t sa[6], const *vendor_ie_data_t* *vnd_ie, int rssi)

Function signature for received Vendor-Specific Information Element callback.

Param ctx Context argument, as passed to esp_wifi_set_vendor_ie_cb() when registering callback.
Param type Information element type, based on frame type received.
Param sa Source 802.11 address.
Param vnd_ie Pointer to the vendor specific element data received.
Param rssi Received signal strength indication.

typedef void (*`wifi_csi_cb_t`)(void *ctx, *wifi_csi_info_t* *data)

The RX callback function of Channel State Information(CSI) data.

```
Each time a CSI data is received, the callback function will be called.
```

Param ctx context argument, passed to esp_wifi_set_csi_rx_cb() when registering callback function.
Param data CSI data received. The memory that it points to will be deallocated after callback function returns.

Header File

- components/esp_wifi/include/esp_wifi_types.h

Unions

union `wifi_config_t`

#include <esp_wifi_types.h> Configuration data for ESP32 AP or STA.

The usage of this union (for ap or sta configuration) is determined by the accompanying interface argument passed to esp_wifi_set_config() or esp_wifi_get_config()

Public Members

wifi_ap_config_t **`ap`**

configuration of AP

wifi_sta_config_t **`sta`**

configuration of STA

Structures

struct **wifi_country_t**

Structure describing WiFi country-based regional restrictions.

Public Members

char **cc**[3]

country code string

uint8_t **schan**

start channel

uint8_t **nchan**

total channel number

int8_t **max_tx_power**

This field is used for getting WiFi maximum transmitting power, call esp_wifi_set_max_tx_power to set the maximum transmitting power.

wifi_country_policy_t **policy**

country policy

struct **wifi_active_scan_time_t**

Range of active scan times per channel.

Public Members

uint32_t **min**

minimum active scan time per channel, units: millisecond

uint32_t **max**

maximum active scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct **wifi_scan_time_t**

Aggregate of active & passive scan time per channel.

Public Members

wifi_active_scan_time_t **active**

active scan time per channel, units: millisecond.

uint32_t **passive**

passive scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct **wifi_scan_config_t**

Parameters for an SSID scan.

Public Members

uint8_t ***ssid**

SSID of AP

uint8_t ***bssid**

MAC address of AP

uint8_t **channel**

channel, scan the specific channel

bool **show_hidden**

enable to scan AP whose SSID is hidden

wifi_scan_type_t **scan_type**

scan type, active or passive

wifi_scan_time_t **scan_time**

scan time per channel

uint8_t **home_chan_dwell_time**

time spent at home channel between scanning consecutive channels.

struct **wifi_ap_record_t**

Description of a WiFi AP.

Public Members

uint8_t **bssid**[6]

MAC address of AP

uint8_t **ssid**[33]

SSID of AP

uint8_t **primary**

channel of AP

wifi_second_chan_t **second**

secondary channel of AP

int8_t **rssi**

signal strength of AP. Note that in some rare cases where signal strength is very strong, rssi values can be slightly positive

wifi_auth_mode_t **authmode**

authmode of AP

wifi_cipher_type_t **pairwise_cipher**

pairwise cipher of AP

wifi_cipher_type_t **group_cipher**

group cipher of AP

wifi_ant_t **ant**

antenna used to receive beacon from AP

uint32_t **phy_11b**

bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g**

bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n**

bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_lr**

bit: 3 flag to identify if low rate is enabled or not

uint32_t **wps**

bit: 4 flag to identify if WPS is supported or not

uint32_t **ftm_responder**

bit: 5 flag to identify if FTM is supported in responder mode

uint32_t **ftm_initiator**

bit: 6 flag to identify if FTM is supported in initiator mode

uint32_t **reserved**

bit: 7..31 reserved

wifi_country_t **country**

country information of AP

struct **wifi_scan_threshold_t**

Structure describing parameters for a WiFi fast scan.

Public Members

int8_t **rssi**

The minimum rssi to accept in the fast scan mode

wifi_auth_mode_t **authmode**

The weakest authmode to accept in the fast scan mode Note: Incase this value is not set and password is set as per WPA2 standards(password len >= 8), it will be defaulted to WPA2 and device won't connect to deprecated WEP/WPA networks. Please set authmode threshold as WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK to connect to WEP/WPA networks

struct **wifi_pmf_config_t**

Configuration structure for Protected Management Frame

Public Members

bool **capable**

Deprecated variable. Device will always connect in PMF mode if other device also advertizes PMF capability.

bool **required**

Advertizes that Protected Management Frame is required. Device will not associate to non-PMF capable devices.

struct **wifi_ap_config_t**

Soft-AP configuration settings for the ESP32.

Public Members

uint8_t **ssid**[32]

SSID of ESP32 soft-AP. If ssid_len field is 0, this must be a Null terminated string. Otherwise, length is set according to ssid_len.

uint8_t **password**[64]

Password of ESP32 soft-AP.

uint8_t **ssid_len**

Optional length of SSID field.

uint8_t **channel**

Channel of soft-AP

wifi_auth_mode_t **authmode**

Auth mode of soft-AP. Do not support AUTH_WEP, AUTH_WAPI_PSK and AUTH_OWE in soft-AP mode. When the auth mode is set to WPA2_PSK, WPA2_WPA3_PSK or WPA3_PSK, the pairwise cipher will be overwritten with WIFI_CIPHER_TYPE_CCMP.

uint8_t **ssid_hidden**

Broadcast SSID or not, default 0, broadcast the SSID

uint8_t **max_connection**

Max number of stations allowed to connect in

uint16_t **beacon_interval**

Beacon interval which should be multiples of 100. Unit: TU(time unit, 1 TU = 1024 us). Range: 100 ~ 60000. Default value: 100

wifi_cipher_type_t **pairwise_cipher**

Pairwise cipher of SoftAP, group cipher will be derived using this. Cipher values are valid starting from WIFI_CIPHER_TYPE_TKIP, enum values before that will be considered as invalid and default cipher suites(TKIP+CCMP) will be used. Valid cipher suites in softAP mode are WIFI_CIPHER_TYPE_TKIP, WIFI_CIPHER_TYPE_CCMP and WIFI_CIPHER_TYPE_TKIP_CCMP.

bool **ftm_responder**

Enable FTM Responder mode

wifi_pmf_config_t **pmf_cfg**

Configuration for Protected Management Frame

struct **wifi_sta_config_t**

STA configuration settings for the ESP32.

Public Members

uint8_t **ssid**[32]

SSID of target AP.

uint8_t **password**[64]

Password of target AP.

wifi_scan_method_t **scan_method**

do all channel scan or fast scan

bool **bssid_set**

whether set MAC address of target AP or not. Generally, station_config.bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

uint8_t **bssid**[6]

MAC address of target AP

uint8_t **channel**

channel of target AP. Set to 1~13 to scan starting from the specified channel before connecting to AP. If the channel of AP is unknown, set it to 0.

uint16_t **listen_interval**

Listen interval for ESP32 station to receive beacon when WIFI_PS_MAX_MODEM is set. Units: AP beacon intervals. Defaults to 3 if set to 0.

wifi_sort_method_t **sort_method**

sort the connect AP in the list by rssi or security mode

wifi_scan_threshold_t **threshold**

When scan_threshold is set, only APs which have an auth mode that is more secure than the selected auth mode and a signal stronger than the minimum RSSI will be used.

wifi_pmf_config_t **pmf_cfg**

Configuration for Protected Management Frame. Will be advertised in RSN Capabilities in RSN IE.

uint32_t **rm_enabled**

Whether Radio Measurements are enabled for the connection

uint32_t **btm_enabled**

Whether BSS Transition Management is enabled for the connection

uint32_t **mbo_enabled**

Whether MBO is enabled for the connection

uint32_t **ft_enabled**

Whether FT is enabled for the connection

uint32_t **owe_enabled**

Whether OWE is enabled for the connection

uint32_t **transition_disable**

Whether to enable transition disable feature

uint32_t **reserved**

Reserved for future feature set

wifi_sae_pwe_method_t **sae_pwe_h2e**

Configuration for SAE PWE derivation method

uint8_t **failure_retry_cnt**

Number of connection retries station will do before moving to next AP. scan_method should be set as WIFI_ALL_CHANNEL_SCAN to use this config. Note: Enabling this may cause connection time to increase incase best AP doesn't behave properly.

struct **wifi_sta_info_t**

Description of STA associated with AP.

Public Members

uint8_t **mac**[6]

mac address

int8_t **rssi**

current average rssi of sta connected

uint32_t **phy_11b**

bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g**

bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n**

bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_lr**

bit: 3 flag to identify if low rate is enabled or not

uint32_t **is_mesh_child**

bit: 4 flag to identify mesh child

uint32_t **reserved**

bit: 5..31 reserved

struct **wifi_sta_list_t**

List of stations associated with the ESP32 Soft-AP.

Public Members

wifi_sta_info_t **sta**[ESP_WIFI_MAX_CONN_NUM]

station list

int **num**

number of stations in the list (other entries are invalid)

struct **vendor_ie_data_t**

Vendor Information Element header.

The first bytes of the Information Element will match this header. Payload follows.

Public Members

uint8_t **element_id**

Should be set to WIFI_VENDOR_IE_ELEMENT_ID (0xDD)

uint8_t **length**

Length of all bytes in the element data following this field. Minimum 4.

uint8_t **vendor_oui**[3]

Vendor identifier (OUI).

uint8_t **vendor_oui_type**

Vendor-specific OUI type.

uint8_t **payload**[0]

Payload. Length is equal to value in ‘length’ field, minus 4.

struct **wifi_pkt_rx_ctrl_t**

Received packet radio metadata header, this is the common header at the beginning of all promiscuous mode RX callback buffers.

Public Members

signed **rssi**

Received Signal Strength Indicator(RSSI) of packet. unit: dBm

unsigned **rate**

PHY rate encoding of the packet. Only valid for non HT(11bg) packet

unsigned **__pad0__**

reserved

unsigned **sig_mode**

0: non HT(11bg) packet; 1: HT(11n) packet; 3: VHT(11ac) packet

unsigned **__pad1__**

reserved

unsigned **mcs**

Modulation Coding Scheme. If is HT(11n) packet, shows the modulation, range from 0 to 76(MSC0 ~ MCS76)

unsigned **cwb**

Channel Bandwidth of the packet. 0: 20MHz; 1: 40MHz

unsigned **__pad2__**

reserved

unsigned **smoothing**

reserved

unsigned **not_sounding**

reserved

unsigned **__pad3__**

reserved

unsigned **aggregation**

Aggregation. 0: MPDU packet; 1: AMPDU packet

unsigned **stbc**

Space Time Block Code(STBC). 0: non STBC packet; 1: STBC packet

unsigned **fec_coding**

Flag is set for 11n packets which are LDPC

unsigned **sgi**

Short Guide Interval(SGI). 0: Long GI; 1: Short GI

signed **noise_floor**

noise floor of Radio Frequency Module(RF). unit: dBm

unsigned **ampdu_cnt**

ampdu cnt

unsigned **channel**

primary channel on which this packet is received

unsigned **secondary_channel**

secondary channel on which this packet is received. 0: none; 1: above; 2: below

unsigned **__pad4__**

reserved

unsigned **timestamp**

timestamp. The local time when this packet is received. It is precise only if modem sleep or light sleep is not enabled. unit: microsecond

unsigned **__pad5__**

reserved

unsigned **__pad6__**

reserved

unsigned **ant**

antenna number from which this packet is received. 0: WiFi antenna 0; 1: WiFi antenna 1

unsigned **sig_len**

length of packet including Frame Check Sequence(FCS)

unsigned **__pad7__**

reserved

unsigned **rx_state**

state of the packet. 0: no error; others: error numbers which are not public

struct **wifi_promiscuous_pkt_t**

Payload passed to ‘buf’ parameter of promiscuous mode RX callback.

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**

metadata header

uint8_t **payload**[0]

Data or management payload. Length of payload is described by rx_ctrl.sig_len. Type of content determined by packet type argument of callback.

struct **wifi_promiscuous_filter_t**

Mask for filtering different packet types in promiscuous mode.

Public Members

uint32_t **filter_mask**

OR of one or more filter values WIFI_PROMIS_FILTER_*

struct **wifi_csi_config_t**

Channel state information(CSI) configuration type.

Public Members

bool **lltf_en**

enable to receive legacy long training field(lltf) data. Default enabled

bool **htltf_en**

enable to receive HT long training field(htltf) data. Default enabled

bool **stbc_htltf2_en**

enable to receive space time block code HT long training field(stbc-htltf2) data. Default enabled

bool **ltf_merge_en**

enable to generate htlft data by averaging lltf and ht_ltf data when receiving HT packet. Otherwise, use ht_ltf data directly. Default enabled

bool **channel_filter_en**

enable to turn on channel filter to smooth adjacent sub-carrier. Disable it to keep independence of adjacent sub-carrier. Default enabled

bool **manu_scale**

manually scale the CSI data by left shifting or automatically scale the CSI data. If set true, please set the shift bits. false: automatically. true: manually. Default false

uint8_t **shift**

manually left shift bits of the scale of the CSI data. The range of the left shift bits is 0~15

bool **dump_ack_en**

enable to dump 802.11 ACK frame, default disabled

struct **wifi_csi_info_t**

CSI data type.

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**

received packet radio metadata header of the CSI data

uint8_t **mac**[6]

source MAC address of the CSI data

uint8_t **dmac**[6]

destination MAC address of the CSI data

bool **first_word_invalid**

first four bytes of the CSI data is invalid or not

int8_t ***buf**

buffer of CSI data

uint16_t **len**

length of CSI data

struct **wifi_ant_gpio_t**

WiFi GPIO configuration for antenna selection.

Public Members

uint8_t **gpio_select**

Whether this GPIO is connected to external antenna switch

uint8_t **gpio_num**

The GPIO number that connects to external antenna switch

struct **wifi_ant_gpio_config_t**

WiFi GPIOs configuration for antenna selection.

Public Members

wifi_ant_gpio_t **gpio_cfg**[4]

The configurations of GPIOs that connect to external antenna switch

struct **wifi_ant_config_t**

WiFi antenna configuration.

Public Members

wifi_ant_mode_t **rx_ant_mode**

WiFi antenna mode for receiving

wifi_ant_t **rx_ant_default**

Default antenna mode for receiving, it’ s ignored if rx_ant_mode is not WIFI_ANT_MODE_AUTO

wifi_ant_mode_t **tx_ant_mode**

WiFi antenna mode for transmission, it can be set to WIFI_ANT_MODE_AUTO only if rx_ant_mode is set to WIFI_ANT_MODE_AUTO

uint8_t **enabled_ant0**

Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT0

uint8_t **enabled_ant1**

Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT1

struct **wifi_action_tx_req_t**

Action Frame Tx Request.

Public Members

wifi_interface_t **ifx**

WiFi interface to send request to

uint8_t **dest_mac**[6]

Destination MAC address

bool **no_ack**

Indicates no ack required

wifi_action_rx_cb_t **rx_cb**

Rx Callback to receive any response

uint32_t **data_len**

Length of the appended Data

uint8_t **data**[0]

Appended Data payload

struct **wifi_ftm_initiator_cfg_t**

FTM Initiator configuration.

Public Members

uint8_t **resp_mac**[6]

MAC address of the FTM Responder

uint8_t **channel**

Primary channel of the FTM Responder

uint8_t **frm_count**

No. of FTM frames requested in terms of 4 or 8 bursts (allowed values - 0(No pref), 16, 24, 32, 64)

uint16_t **burst_period**

Requested period between FTM bursts in 100’ s of milliseconds (allowed values 0(No pref) - 100)

bool **use_get_report_api**

True - Using esp_wifi_ftm_get_report to get FTM report, False - Using ftm_report_data from WIFI_EVENT_FTM_REPORT to get FTM report

struct **wifi_event_sta_scan_done_t**

Argument structure for WIFI_EVENT_SCAN_DONE event

Public Members

uint32_t **status**

status of scanning APs: 0 —success, 1 - failure

uint8_t **number**

number of scan results

uint8_t **scan_id**

scan sequence number, used for block scan

struct **wifi_event_sta_connected_t**

Argument structure for WIFI_EVENT_STA_CONNECTED event

Public Members

uint8_t **ssid**[32]

SSID of connected AP

uint8_t **ssid_len**

SSID length of connected AP

uint8_t **bssid**[6]

BSSID of connected AP

uint8_t **channel**

channel of connected AP

wifi_auth_mode_t **authmode**

authentication mode used by AP

struct **wifi_event_sta_disconnected_t**

Argument structure for WIFI_EVENT_STA_DISCONNECTED event

Public Members

uint8_t **ssid**[32]

SSID of disconnected AP

uint8_t **ssid_len**

SSID length of disconnected AP

uint8_t **bssid**[6]

BSSID of disconnected AP

uint8_t **reason**

reason of disconnection

int8_t **rssi**

rssi of disconnection

struct **wifi_event_sta_authmode_change_t**

Argument structure for WIFI_EVENT_STA_AUTHMODE_CHANGE event

Public Members

wifi_auth_mode_t **old_mode**

the old auth mode of AP

wifi_auth_mode_t **new_mode**

the new auth mode of AP

struct **wifi_event_sta_wps_er_pin_t**

Argument structure for WIFI_EVENT_STA_WPS_ER_PIN event

Public Members

uint8_t **pin_code**[8]

PIN code of station in enrollee mode

struct **wifi_event_sta_wps_er_success_t**

Argument structure for WIFI_EVENT_STA_WPS_ER_SUCCESS event

Public Members

uint8_t **ap_cred_cnt**

Number of AP credentials received

uint8_t **ssid**[MAX_SSID_LEN]

SSID of AP

uint8_t **passphrase**[MAX_PASSPHRASE_LEN]

Passphrase for the AP

struct *wifi_event_sta_wps_er_success_t*::[anonymous] **ap_cred**[MAX_WPS_AP_CRED]

All AP credentials received from WPS handshake

struct **wifi_event_ap_staconnected_t**

Argument structure for WIFI_EVENT_AP_STACONNECTED event

Public Members

uint8_t **mac**[6]

MAC address of the station connected to ESP32 soft-AP

uint8_t **aid**

the aid that ESP32 soft-AP gives to the station connected to

bool **is_mesh_child**

flag to identify mesh child

struct **wifi_event_ap_stadisconnected_t**

Argument structure for WIFI_EVENT_AP_STADISCONNECTED event

Public Members

uint8_t **mac**[6]

MAC address of the station disconnects to ESP32 soft-AP

uint8_t **aid**

the aid that ESP32 soft-AP gave to the station disconnects to

bool **is_mesh_child**

flag to identify mesh child

struct **wifi_event_ap_probe_req_rx_t**

Argument structure for WIFI_EVENT_AP_PROBEREQRECVED event

Public Members

int **rssi**

Received probe request signal strength

uint8_t **mac**[6]

MAC address of the station which send probe request

struct **wifi_event_bss_rssi_low_t**

Argument structure for WIFI_EVENT_STA_BSS_RSSI_LOW event

Public Members

int32_t **rssi**

RSSI value of bss

struct **wifi_ftm_report_entry_t**

Argument structure for

Public Members

uint8_t **dlog_token**

Dialog Token of the FTM frame

int8_t **rssi**

RSSI of the FTM frame received

uint32_t **rtt**

Round Trip Time in pSec with a peer

uint64_t **t1**

Time of departure of FTM frame from FTM Responder in pSec

uint64_t **t2**

Time of arrival of FTM frame at FTM Initiator in pSec

uint64_t **t3**

Time of departure of ACK from FTM Initiator in pSec

uint64_t **t4**

Time of arrival of ACK at FTM Responder in pSec

struct **wifi_event_ftm_report_t**

Argument structure for WIFI_EVENT_FTM_REPORT event

Public Members

uint8_t **peer_mac**[6]

MAC address of the FTM Peer

wifi_ftm_status_t **status**

Status of the FTM operation

uint32_t **rtt_raw**

Raw average Round-Trip-Time with peer in Nano-Seconds

uint32_t **rtt_est**

Estimated Round-Trip-Time with peer in Nano-Seconds

uint32_t **dist_est**

Estimated one-way distance in Centi-Meters

wifi_ftm_report_entry_t ***ftm_report_data**

Pointer to FTM Report, should be freed after use. Note: Highly recommended to use API esp_wifi_ftm_get_report to get the report instead of using this

uint8_t **ftm_report_num_entries**

Number of entries in the FTM Report data

struct **wifi_event_action_tx_status_t**

Argument structure for WIFI_EVENT_ACTION_TX_STATUS event

Public Members

wifi_interface_t **ifx**

WiFi interface to send request to

uint32_t **context**

Context to identify the request

uint8_t **da**[6]

Destination MAC address

uint8_t **status**

Status of the operation

struct **wifi_event_roc_done_t**

Argument structure for WIFI_EVENT_ROC_DONE event

Public Members

uint32_t **context**

Context to identify the request

struct **wifi_event_ap_wps_rg_pin_t**

Argument structure for WIFI_EVENT_AP_WPS_RG_PIN event

Public Members

uint8_t **pin_code**[8]

PIN code of station in enrollee mode

struct **wifi_event_ap_wps_rg_fail_reason_t**

Argument structure for WIFI_EVENT_AP_WPS_RG_FAILED event

Public Members

wps_fail_reason_t **reason**

WPS failure reason wps_fail_reason_t

uint8_t **peer_macaddr**[6]

Enrollee mac address

struct **wifi_event_ap_wps_rg_success_t**

Argument structure for WIFI_EVENT_AP_WPS_RG_SUCCESS event

Public Members

uint8_t **peer_macaddr**[6]

Enrollee mac address

Macros

WIFI_OFFCHAN_TX_REQ

WIFI_OFFCHAN_TX_CANCEL

WIFI_ROC_REQ

WIFI_ROC_CANCEL

WIFI_PROTOCOL_11B

WIFI_PROTOCOL_11G

WIFI_PROTOCOL_11N

WIFI_PROTOCOL_LR

ESP_WIFI_MAX_CONN_NUM

max number of stations which can connect to ESP32/ESP32S3/ESP32S2 soft-AP

WIFI_VENDOR_IE_ELEMENT_ID

WIFI_PROMIS_FILTER_MASK_ALL

filter all packets

WIFI_PROMIS_FILTER_MASK_MGMT

filter the packets with type of WIFI_PKT_MGMT

WIFI_PROMIS_FILTER_MASK_CTRL

filter the packets with type of WIFI_PKT_CTRL

WIFI_PROMIS_FILTER_MASK_DATA

filter the packets with type of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_MISC

filter the packets with type of WIFI_PKT_MISC

WIFI_PROMIS_FILTER_MASK_DATA_MPDU

filter the MPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_DATA_AMPDU

filter the AMPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_FCSFAIL

filter the FCS failed packets, do not open it in general

WIFI_PROMIS_CTRL_FILTER_MASK_ALL

filter all control packets

WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER

filter the control packets with subtype of Control Wrapper

WIFI_PROMIS_CTRL_FILTER_MASK_BAR

filter the control packets with subtype of Block Ack Request

WIFI_PROMIS_CTRL_FILTER_MASK_BA

filter the control packets with subtype of Block Ack

WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL

filter the control packets with subtype of PS-Poll

WIFI_PROMIS_CTRL_FILTER_MASK_RTS

filter the control packets with subtype of RTS

WIFI_PROMIS_CTRL_FILTER_MASK_CTS

filter the control packets with subtype of CTS

WIFI_PROMIS_CTRL_FILTER_MASK_ACK

filter the control packets with subtype of ACK

WIFI_PROMIS_CTRL_FILTER_MASK_CFEND

filter the control packets with subtype of CF-END

WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK

filter the control packets with subtype of CF-END+CF-ACK

WIFI_EVENT_MASK_ALL

mask all WiFi events

WIFI_EVENT_MASK_NONE

mask none of the WiFi events

WIFI_EVENT_MASK_AP_PROBEREQRECVED

mask SYSTEM_EVENT_AP_PROBEREQRECVED event

MAX_SSID_LEN

MAX_PASSPHRASE_LEN

MAX_WPS_AP_CRED

WIFI_STATIS_BUFFER

WIFI_STATIS_RXTX

WIFI_STATIS_HW

WIFI_STATIS_DIAG

WIFI_STATIS_PS

WIFI_STATIS_ALL

Type Definitions

typedef int (***wifi_action_rx_cb_t**)(uint8_t *hdr, uint8_t *payload, size_t len, uint8_t channel)

The Rx callback function of Action Tx operations.

Param hdr pointer to the IEEE 802.11 Header structure
Param payload pointer to the Payload following 802.11 Header
Param len length of the Payload
Param channel channel number the frame is received on

Enumerations

enum **wifi_mode_t**

Values:

enumerator **WIFI_MODE_NULL**

null mode

enumerator **WIFI_MODE_STA**

WiFi station mode

enumerator **WIFI_MODE_AP**

WiFi soft-AP mode

enumerator **WIFI_MODE_APSTA**

WiFi station + soft-AP mode

enumerator **WIFI_MODE_MAX**

enum **wifi_interface_t**

Values:

enumerator **WIFI_IF_STA**

enumerator **WIFI_IF_AP**

enum **wifi_country_policy_t**

Values:

enumerator **WIFI_COUNTRY_POLICY_AUTO**

Country policy is auto, use the country info of AP to which the station is connected

enumerator **WIFI_COUNTRY_POLICY_MANUAL**

Country policy is manual, always use the configured country info

enum **wifi_auth_mode_t**

Values:

enumerator **WIFI_AUTH_OPEN**

authenticate mode : open

enumerator **WIFI_AUTH_WEP**

authenticate mode : WEP

enumerator **WIFI_AUTH_WPA_PSK**

authenticate mode : WPA_PSK

enumerator **WIFI_AUTH_WPA2_PSK**

authenticate mode : WPA2_PSK

enumerator **WIFI_AUTH_WPA_WPA2_PSK**

authenticate mode : WPA_WPA2_PSK

enumerator **WIFI_AUTH_ENTERPRISE**

authenticate mode : WiFi EAP security

enumerator **WIFI_AUTH_WPA2_ENTERPRISE**

authenticate mode : WiFi EAP security

enumerator **WIFI_AUTH_WPA3_PSK**

authenticate mode : WPA3_PSK

enumerator **WIFI_AUTH_WPA2_WPA3_PSK**

authenticate mode : WPA2_WPA3_PSK

enumerator **WIFI_AUTH_WAPI_PSK**

authenticate mode : WAPI_PSK

enumerator **WIFI_AUTH_OWE**

authenticate mode : OWE

enumerator **WIFI_AUTH_WPA3_ENT_192**

authenticate mode : WPA3_ENT_SUITE_B_192_BIT

enumerator **WIFI_AUTH_MAX**

enum **wifi_err_reason_t**

Values:

enumerator **WIFI_REASON_UNSPECIFIED**

enumerator **WIFI_REASON_AUTH_EXPIRE**

enumerator **WIFI_REASON_AUTH_LEAVE**

enumerator **WIFI_REASON_ASSOC_EXPIRE**

enumerator **WIFI_REASON_ASSOC_TOOMANY**

enumerator **WIFI_REASON_NOT_AUTHED**

enumerator **WIFI_REASON_NOT_ASSOCED**

enumerator **WIFI_REASON_ASSOC_LEAVE**

enumerator **WIFI_REASON_ASSOC_NOT_AUTHED**

enumerator **WIFI_REASON_DISASSOC_PWRCAP_BAD**

enumerator **WIFI_REASON_DISASSOC_SUPCHAN_BAD**

enumerator **WIFI_REASON_BSS_TRANSITION_DISASSOC**

enumerator **WIFI_REASON_IE_INVALID**

enumerator **WIFI_REASON_MIC_FAILURE**

enumerator **WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT**

enumerator **WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT**

enumerator **WIFI_REASON_IE_IN_4WAY_DIFFERS**

enumerator **WIFI_REASON_GROUP_CIPHER_INVALID**

enumerator **WIFI_REASON_PAIRWISE_CIPHER_INVALID**

enumerator **WIFI_REASON_AKMP_INVALID**

enumerator **WIFI_REASON_UNSUPP_RSN_IE_VERSION**

enumerator **WIFI_REASON_INVALID_RSN_IE_CAP**

enumerator **WIFI_REASON_802_1X_AUTH_FAILED**

enumerator **WIFI_REASON_CIPHER_SUITE_REJECTED**

enumerator **WIFI_REASON_TDLS_PEER_UNREACHABLE**

enumerator **WIFI_REASON_TDLS_UNSPECIFIED**

enumerator **WIFI_REASON_SSP_REQUESTED_DISASSOC**

enumerator **WIFI_REASON_NO_SSP_ROAMING_AGREEMENT**

enumerator **WIFI_REASON_BAD_CIPHER_OR_AKM**

enumerator **WIFI_REASON_NOT_AUTHORIZED_THIS_LOCATION**

enumerator **WIFI_REASON_SERVICE_CHANGE_PERCLUDES_TS**

enumerator **WIFI_REASON_UNSPECIFIED_QOS**

enumerator **WIFI_REASON_NOT_ENOUGH_BANDWIDTH**

enumerator **WIFI_REASON_MISSING_ACKS**

enumerator **WIFI_REASON_EXCEEDED_TXOP**

enumerator **WIFI_REASON_STA_LEAVING**

enumerator **WIFI_REASON_END_BA**

enumerator **WIFI_REASON_UNKNOWN_BA**

enumerator **WIFI_REASON_TIMEOUT**

enumerator **WIFI_REASON_PEER_INITIATED**

enumerator **WIFI_REASON_AP_INITIATED**

enumerator **WIFI_REASON_INVALID_FT_ACTION_FRAME_COUNT**

enumerator **WIFI_REASON_INVALID_PMKID**

enumerator **WIFI_REASON_INVALID_MDE**

enumerator **WIFI_REASON_INVALID_FTE**

enumerator **WIFI_REASON_TRANSMISSION_LINK_ESTABLISH_FAILED**

enumerator **WIFI_REASON_ALTERATIVE_CHANNEL_OCCUPIED**

enumerator **WIFI_REASON_BEACON_TIMEOUT**

enumerator **WIFI_REASON_NO_AP_FOUND**

enumerator **WIFI_REASON_AUTH_FAIL**

enumerator **WIFI_REASON_ASSOC_FAIL**

enumerator **WIFI_REASON_HANDSHAKE_TIMEOUT**

enumerator **WIFI_REASON_CONNECTION_FAIL**

enumerator **WIFI_REASON_AP_TSF_RESET**

enumerator **WIFI_REASON_ROAMING**

enumerator **WIFI_REASON_ASSOC_COMEBACK_TIME_TOO_LONG**

enumerator **WIFI_REASON_SA_QUERY_TIMEOUT**

enum **wifi_second_chan_t**

Values:

enumerator **WIFI_SECOND_CHAN_NONE**

the channel width is HT20

enumerator **WIFI_SECOND_CHAN_ABOVE**

the channel width is HT40 and the secondary channel is above the primary channel

enumerator **WIFI_SECOND_CHAN_BELOW**

the channel width is HT40 and the secondary channel is below the primary channel

enum **wifi_scan_type_t**

Values:

enumerator **WIFI_SCAN_TYPE_ACTIVE**

active scan

enumerator **WIFI_SCAN_TYPE_PASSIVE**

passive scan

enum **wifi_cipher_type_t**

Values:

enumerator **WIFI_CIPHER_TYPE_NONE**

the cipher type is none

enumerator **WIFI_CIPHER_TYPE_WEP40**

the cipher type is WEP40

enumerator **WIFI_CIPHER_TYPE_WEP104**

the cipher type is WEP104

enumerator **WIFI_CIPHER_TYPE_TKIP**

the cipher type is TKIP

enumerator **WIFI_CIPHER_TYPE_CCMP**

the cipher type is CCMP

enumerator **WIFI_CIPHER_TYPE_TKIP_CCMP**

the cipher type is TKIP and CCMP

enumerator **WIFI_CIPHER_TYPE_AES_CMAC128**

the cipher type is AES-CMAC-128

enumerator **WIFI_CIPHER_TYPE_SMS4**

the cipher type is SMS4

enumerator **WIFI_CIPHER_TYPE_GCMP**

the cipher type is GCMP

enumerator **WIFI_CIPHER_TYPE_GCMP256**

the cipher type is GCMP-256

enumerator **WIFI_CIPHER_TYPE_AES_GMAC128**

the cipher type is AES-GMAC-128

enumerator **WIFI_CIPHER_TYPE_AES_GMAC256**

the cipher type is AES-GMAC-256

enumerator **WIFI_CIPHER_TYPE_UNKNOWN**

the cipher type is unknown

enum **wifi_ant_t**

WiFi antenna.

Values:

enumerator **WIFI_ANT_ANT0**

WiFi antenna 0

enumerator **WIFI_ANT_ANT1**

WiFi antenna 1

enumerator **WIFI_ANT_MAX**

Invalid WiFi antenna

enum **wifi_scan_method_t**

Values:

enumerator **WIFI_FAST_SCAN**

Do fast scan, scan will end after find SSID match AP

enumerator **WIFI_ALL_CHANNEL_SCAN**

All channel scan, scan will end after scan all the channel

enum **wifi_sort_method_t**

Values:

enumerator **WIFI_CONNECT_AP_BY_SIGNAL**

Sort match AP in scan list by RSSI

enumerator **WIFI_CONNECT_AP_BY_SECURITY**

Sort match AP in scan list by security mode

enum **wifi_ps_type_t**

Values:

enumerator **WIFI_PS_NONE**

No power save

enumerator **WIFI_PS_MIN_MODEM**

Minimum modem power saving. In this mode, station wakes up to receive beacon every DTIM period

enumerator **WIFI_PS_MAX_MODEM**

Maximum modem power saving. In this mode, interval to receive beacons is determined by the listen_interval parameter in *wifi_sta_config_t*

enum **wifi_bandwidth_t**

Values:

enumerator **WIFI_BW_HT20**

enumerator **WIFI_BW_HT40**

enum **wifi_sae_pwe_method_t**

Configuration for SAE PWE derivation

Values:

enumerator **WPA3_SAE_PWE_UNSPECIFIED**

enumerator **WPA3_SAE_PWE_HUNT_AND_PECK**

enumerator **WPA3_SAE_PWE_HASH_TO_ELEMENT**

enumerator **WPA3_SAE_PWE_BOTH**

enum **wifi_storage_t**

Values:

enumerator **WIFI_STORAGE_FLASH**

all configuration will store in both memory and flash

enumerator **WIFI_STORAGE_RAM**

all configuration will only store in the memory

enum **wifi_vendor_ie_type_t**

Vendor Information Element type.

Determines the frame type that the IE will be associated with.

Values:

enumerator **WIFI_VND_IE_TYPE_BEACON**

enumerator **WIFI_VND_IE_TYPE_PROBE_REQ**

enumerator **WIFI_VND_IE_TYPE_PROBE_RESP**

enumerator **WIFI_VND_IE_TYPE_ASSOC_REQ**

enumerator **WIFI_VND_IE_TYPE_ASSOC_RESP**

enum **wifi_vendor_ie_id_t**

Vendor Information Element index.

Each IE type can have up to two associated vendor ID elements.

Values:

enumerator **WIFI_VND_IE_ID_0**

enumerator **WIFI_VND_IE_ID_1**

enum **wifi_phy_mode_t**

Operation Phymode.

Values:

enumerator **WIFI_PHY_MODE_LR**

PHY mode for Low Rate

enumerator **WIFI_PHY_MODE_11B**

PHY mode for 11b

enumerator **WIFI_PHY_MODE_11G**

PHY mode for 11g

enumerator **WIFI_PHY_MODE_HT20**

PHY mode for Bandwidth HT20

enumerator **WIFI_PHY_MODE_HT40**

PHY mode for Bandwidth HT40

enumerator **WIFI_PHY_MODE_HE20**

PHY mode for Bandwidth HE20

enum **wifi_promiscuous_pkt_type_t**

Promiscuous frame type.

Passed to promiscuous mode RX callback to indicate the type of parameter in the buffer.

Values:

enumerator **WIFI_PKT_MGMT**

Management frame, indicates ‘buf’ argument is *wifi_promiscuous_pkt_t*

enumerator **WIFI_PKT_CTRL**

Control frame, indicates ‘buf’ argument is *wifi_promiscuous_pkt_t*

enumerator **WIFI_PKT_DATA**

Data frame, indiciates ‘buf’ argument is *wifi_promiscuous_pkt_t*

enumerator **WIFI_PKT_MISC**

Other type, such as MIMO etc. ‘buf’ argument is *wifi_promiscuous_pkt_t* but the payload is zero length.

enum **wifi_ant_mode_t**

WiFi antenna mode.

Values:

enumerator **WIFI_ANT_MODE_ANT0**

Enable WiFi antenna 0 only

enumerator **WIFI_ANT_MODE_ANT1**

Enable WiFi antenna 1 only

enumerator **WIFI_ANT_MODE_AUTO**

Enable WiFi antenna 0 and 1, automatically select an antenna

enumerator **WIFI_ANT_MODE_MAX**

Invalid WiFi enabled antenna

enum **wifi_phy_rate_t**

WiFi PHY rate encodings.

Values:

enumerator **WIFI_PHY_RATE_1M_L**

1 Mbps with long preamble

enumerator **WIFI_PHY_RATE_2M_L**

2 Mbps with long preamble

enumerator **WIFI_PHY_RATE_5M_L**

5.5 Mbps with long preamble

enumerator **WIFI_PHY_RATE_11M_L**

11 Mbps with long preamble

enumerator **WIFI_PHY_RATE_2M_S**

2 Mbps with short preamble

enumerator **WIFI_PHY_RATE_5M_S**

5.5 Mbps with short preamble

enumerator **WIFI_PHY_RATE_11M_S**

11 Mbps with short preamble

enumerator **WIFI_PHY_RATE_48M**

48 Mbps

enumerator **WIFI_PHY_RATE_24M**

24 Mbps

enumerator **WIFI_PHY_RATE_12M**

12 Mbps

enumerator **WIFI_PHY_RATE_6M**

6 Mbps

enumerator **WIFI_PHY_RATE_54M**

54 Mbps

enumerator **WIFI_PHY_RATE_36M**

36 Mbps

enumerator **WIFI_PHY_RATE_18M**

18 Mbps

enumerator **WIFI_PHY_RATE_9M**

9 Mbps

enumerator **WIFI_PHY_RATE_MCS0_LGI**

MCS0 with long GI, 6.5 Mbps for 20MHz, 13.5 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS1_LGI**

MCS1 with long GI, 13 Mbps for 20MHz, 27 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS2_LGI**

MCS2 with long GI, 19.5 Mbps for 20MHz, 40.5 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS3_LGI**

MCS3 with long GI, 26 Mbps for 20MHz, 54 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS4_LGI**

MCS4 with long GI, 39 Mbps for 20MHz, 81 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS5_LGI**

MCS5 with long GI, 52 Mbps for 20MHz, 108 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS6_LGI**

MCS6 with long GI, 58.5 Mbps for 20MHz, 121.5 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS7_LGI**

MCS7 with long GI, 65 Mbps for 20MHz, 135 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS0_SGI**

MCS0 with short GI, 7.2 Mbps for 20MHz, 15 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS1_SGI**

MCS1 with short GI, 14.4 Mbps for 20MHz, 30 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS2_SGI**

MCS2 with short GI, 21.7 Mbps for 20MHz, 45 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS3_SGI**

MCS3 with short GI, 28.9 Mbps for 20MHz, 60 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS4_SGI**

MCS4 with short GI, 43.3 Mbps for 20MHz, 90 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS5_SGI**

MCS5 with short GI, 57.8 Mbps for 20MHz, 120 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS6_SGI**

MCS6 with short GI, 65 Mbps for 20MHz, 135 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_MCS7_SGI**

MCS7 with short GI, 72.2 Mbps for 20MHz, 150 Mbps for 40MHz

enumerator **WIFI_PHY_RATE_LORA_250K**

250 Kbps

enumerator **WIFI_PHY_RATE_LORA_500K**

500 Kbps

enumerator **WIFI_PHY_RATE_MAX**

enum **wifi_event_t**

WiFi event declarations

Values:

enumerator **WIFI_EVENT_WIFI_READY**

ESP32 WiFi ready

enumerator **WIFI_EVENT_SCAN_DONE**

ESP32 finish scanning AP

enumerator **WIFI_EVENT_STA_START**

ESP32 station start

enumerator **WIFI_EVENT_STA_STOP**

ESP32 station stop

enumerator **WIFI_EVENT_STA_CONNECTED**

ESP32 station connected to AP

enumerator **WIFI_EVENT_STA_DISCONNECTED**

ESP32 station disconnected from AP

enumerator **WIFI_EVENT_STA_AUTHMODE_CHANGE**

the auth mode of AP connected by ESP32 station changed

enumerator **WIFI_EVENT_STA_WPS_ER_SUCCESS**

ESP32 station wps succeeds in enrollee mode

enumerator **WIFI_EVENT_STA_WPS_ER_FAILED**
ESP32 station wps fails in enrollee mode

enumerator **WIFI_EVENT_STA_WPS_ER_TIMEOUT**
ESP32 station wps timeout in enrollee mode

enumerator **WIFI_EVENT_STA_WPS_ER_PIN**
ESP32 station wps pin code in enrollee mode

enumerator **WIFI_EVENT_STA_WPS_ER_PBC_OVERLAP**
ESP32 station wps overlap in enrollee mode

enumerator **WIFI_EVENT_AP_START**
ESP32 soft-AP start

enumerator **WIFI_EVENT_AP_STOP**
ESP32 soft-AP stop

enumerator **WIFI_EVENT_AP_STACONNECTED**
a station connected to ESP32 soft-AP

enumerator **WIFI_EVENT_AP_STADISCONNECTED**
a station disconnected from ESP32 soft-AP

enumerator **WIFI_EVENT_AP_PROBEREQRECVED**
Receive probe request packet in soft-AP interface

enumerator **WIFI_EVENT_FTM_REPORT**
Receive report of FTM procedure

enumerator **WIFI_EVENT_STA_BSS_RSSI_LOW**
AP’ s RSSI crossed configured threshold

enumerator **WIFI_EVENT_ACTION_TX_STATUS**
Status indication of Action Tx operation

enumerator **WIFI_EVENT_ROC_DONE**
Remain-on-Channel operation complete

enumerator **WIFI_EVENT_STA_BEACON_TIMEOUT**
ESP32 station beacon timeout

enumerator **WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START**
ESP32 connectionless module wake interval start

enumerator **WIFI_EVENT_AP_WPS_RG_SUCCESS**
Soft-AP wps succeeds in registrar mode

enumerator **WIFI_EVENT_AP_WPS_RG_FAILED**

Soft-AP wps fails in registrar mode

enumerator **WIFI_EVENT_AP_WPS_RG_TIMEOUT**

Soft-AP wps timeout in registrar mode

enumerator **WIFI_EVENT_AP_WPS_RG_PIN**

Soft-AP wps pin code in registrar mode

enumerator **WIFI_EVENT_AP_WPS_RG_PBC_OVERLAP**

Soft-AP wps overlap in registrar mode

enumerator **WIFI_EVENT_MAX**

Invalid WiFi event ID

enum **wifi_event_sta_wps_fail_reason_t**

Argument structure for WIFI_EVENT_STA_WPS_ER_FAILED event

Values:

enumerator **WPS_FAIL_REASON_NORMAL**

ESP32 WPS normal fail reason

enumerator **WPS_FAIL_REASON_RECV_M2D**

ESP32 WPS receive M2D frame

enumerator **WPS_FAIL_REASON_MAX**

enum **wifi_ftm_status_t**

FTM operation status types.

Values:

enumerator **FTM_STATUS_SUCCESS**

FTM exchange is successful

enumerator **FTM_STATUS_UNSUPPORTED**

Peer does not support FTM

enumerator **FTM_STATUS_CONF_REJECTED**

Peer rejected FTM configuration in FTM Request

enumerator **FTM_STATUS_NO_RESPONSE**

Peer did not respond to FTM Requests

enumerator **FTM_STATUS_FAIL**

Unknown error during FTM exchange

enumerator **FTM_STATUS_NO_VALID_MSMT**

FTM session did not result in any valid measurements

enumerator **FTM_STATUS_USER_TERM**

User triggered termination

enum **wps_fail_reason_t**

Values:

enumerator **WPS_AP_FAIL_REASON_NORMAL**

WPS normal fail reason

enumerator **WPS_AP_FAIL_REASON_CONFIG**

WPS failed due to incorrect config

enumerator **WPS_AP_FAIL_REASON_AUTH**

WPS failed during auth

enumerator **WPS_AP_FAIL_REASON_MAX**

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_eap_client.h

Functions

esp_err_t **esp_wifi_sta_enterprise_enable**(void)

Enable EAP authentication(WiFi Enterprise) for the station mode.

This function enables Extensible Authentication Protocol (EAP) authentication for the Wi-Fi station mode. When EAP authentication is enabled, the ESP device will attempt to authenticate with the configured EAP credentials when connecting to a secure Wi-Fi network.

Note: Before calling this function, ensure that the Wi-Fi configuration and EAP credentials (such as username and password) have been properly set using the appropriate configuration APIs.

Returns

- ESP_OK: EAP authentication enabled successfully.
- ESP_ERR_NO_MEM: Failed to enable EAP authentication due to memory allocation failure.

esp_err_t **esp_wifi_sta_enterprise_disable**(void)

Disable EAP authentication(WiFi Enterprise) for the station mode.

This function disables Extensible Authentication Protocol (EAP) authentication for the Wi-Fi station mode. When EAP authentication is disabled, the ESP device will not attempt to authenticate using EAP credentials when connecting to a secure Wi-Fi network.

Note: Disabling EAP authentication may cause the device to connect to the Wi-Fi network using other available authentication methods, if configured using esp_wifi_set_config().

Returns

- ESP_OK: EAP authentication disabled successfully.
- ESP_ERR_INVALID_STATE: EAP client is in an invalid state for disabling.

esp_err_t **esp_eap_client_set_identity**(const unsigned char *identity, int len)

Set identity for PEAP/TTLS authentication method.

This function sets the identity to be used during PEAP/TTLS authentication.

Parameters
- **identity** –**[in]** Pointer to the identity data.
- **len** –**[in]** Length of the identity data (limited to 1~127 bytes).

Returns
- ESP_OK: The identity was set successfully.
- ESP_ERR_INVALID_ARG: Invalid argument (len <= 0 or len >= 128).
- ESP_ERR_NO_MEM: Memory allocation failure.

void **esp_eap_client_clear_identity**(void)

Clear the previously set identity for PEAP/TTLS authentication.

This function clears the identity that was previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured identity during the authentication process.

esp_err_t **esp_eap_client_set_username**(const unsigned char *username, int len)

Set username for PEAP/TTLS authentication method.

This function sets the username to be used during PEAP/TTLS authentication.

Parameters
- **username** –**[in]** Pointer to the username data.
- **len** –**[in]** Length of the username data (limited to 1~127 bytes).

Returns
- ESP_OK: The username was set successfully.
- ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0 or len >= 128).
- ESP_ERR_NO_MEM: Failed due to memory allocation failure.

void **esp_eap_client_clear_username**(void)

Clear username for PEAP/TTLS method.

This function clears the previously set username for the EAP client.

esp_err_t **esp_eap_client_set_password**(const unsigned char *password, int len)

Set password for PEAP/TTLS authentication method.

This function sets the password to be used during PEAP/TTLS authentication.

Parameters
- **password** –**[in]** Pointer to the password data.
- **len** –**[in]** Length of the password data (len > 0).

Returns
- ESP_OK: The password was set successfully.
- ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0).
- ESP_ERR_NO_MEM: Failed due to memory allocation failure.

void **esp_eap_client_clear_password**(void)

Clear password for PEAP/TTLS method.

This function clears the previously set password for the EAP client.

esp_err_t **esp_eap_client_set_new_password**(const unsigned char *new_password, int len)

Set a new password for MSCHAPv2 authentication method.

This function sets the new password to be used during MSCHAPv2 authentication. The new password is used to substitute the old password when an eap-mschapv2 failure request message with error code ERROR_PASSWD_EXPIRED is received.

Parameters
- **new_password** –**[in]** Pointer to the new password data.
- **len** –**[in]** Length of the new password data.

Returns

- ESP_OK: The new password was set successfully.
- ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0).
- ESP_ERR_NO_MEM: Failed due to memory allocation failure.

void **esp_eap_client_clear_new_password**(void)

Clear new password for MSCHAPv2 method.

This function clears the previously set new password for the EAP client.

esp_err_t **esp_eap_client_set_ca_cert**(const unsigned char *ca_cert, int ca_cert_len)

Set CA certificate for EAP authentication.

This function sets the Certificate Authority (CA) certificate to be used during EAP authentication. The CA certificate is passed to the EAP client module through a global pointer.

Parameters

- **ca_cert** –**[in]** Pointer to the CA certificate data.
- **ca_cert_len** –**[in]** Length of the CA certificate data.

Returns

- ESP_OK: The CA certificate was set successfully.

void **esp_eap_client_clear_ca_cert**(void)

Clear the previously set Certificate Authority (CA) certificate for EAP authentication.

This function clears the CA certificate that was previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured CA certificate during the authentication process.

esp_err_t **esp_eap_client_set_certificate_and_key**(const unsigned char *client_cert, int client_cert_len, const unsigned char *private_key, int private_key_len, const unsigned char *private_key_password, int private_key_passwd_len)

Set client certificate and private key for EAP authentication.

This function sets the client certificate and private key to be used during authentication. Optionally, a private key password can be provided for encrypted private keys.

Attention 1. The client certificate, private key, and private key password are provided as pointers to the respective data arrays.

Attention 2. The client_cert, private_key, and private_key_password should be zero-terminated.

Parameters

- **client_cert** –**[in]** Pointer to the client certificate data.
- **client_cert_len** –**[in]** Length of the client certificate data.
- **private_key** –**[in]** Pointer to the private key data.
- **private_key_len** –**[in]** Length of the private key data (limited to 1~4096 bytes).
- **private_key_password** –**[in]** Pointer to the private key password data (optional).
- **private_key_passwd_len** –**[in]** Length of the private key password data (can be 0 for no password).

Returns

- ESP_OK: The certificate, private key, and password (if provided) were set successfully.

void **esp_eap_client_clear_certificate_and_key**(void)

Clear the previously set client certificate and private key for EAP authentication.

This function clears the client certificate and private key that were previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured certificate and private key during the authentication process.

esp_err_t **esp_eap_client_set_disable_time_check**(bool disable)

Set EAP client certificates time check (disable or not).

This function enables or disables the time check for EAP client certificates. When disabled, the certificates' expiration time will not be checked during the authentication process.

Parameters **disable** –**[in]** True to disable EAP client certificates time check, false to enable it.
Returns
- ESP_OK: The EAP client certificates time check setting was updated successfully.

esp_err_t **esp_eap_client_get_disable_time_check**(bool *disable)

Get EAP client certificates time check status.

This function retrieves the current status of the EAP client certificates time check.

Parameters **disable** –**[out]** Pointer to a boolean variable to store the disable status.
Returns
- ESP_OK: The status of EAP client certificates time check was retrieved successfully.

esp_err_t **esp_eap_client_set_ttls_phase2_method**(*esp_eap_ttls_phase2_types* type)

Set EAP-TTLS phase 2 method.

This function sets the phase 2 method to be used during EAP-TTLS authentication.

Parameters **type** –**[in]** The type of phase 2 method to be used (e.g., EAP, MSCHAPv2, MSCHAP, PAP, CHAP).
Returns
- ESP_OK: The EAP-TTLS phase 2 method was set successfully.

esp_err_t **esp_eap_client_set_suiteb_192bit_certification**(bool enable)

Enable or disable Suite-B 192-bit certification checks.

This function enables or disables the 192-bit Suite-B certification checks during EAP-TLS authentication. Suite-B is a set of cryptographic algorithms which generally are considered more secure.

Parameters **enable** –**[in]** True to enable 192-bit Suite-B certification checks, false to disable it.
Returns
- ESP_OK: The 192-bit Suite-B certification checks were set successfully.

esp_err_t **esp_eap_client_set_pac_file**(const unsigned char *pac_file, int pac_file_len)

Set the PAC (Protected Access Credential) file for EAP-FAST authentication.

EAP-FAST requires a PAC file that contains the client' s credentials.

Attention 1. For files read from the file system, length has to be decremented by 1 byte.
Attention 2. Disabling the WPA_MBEDTLS_TLS_CLIENT config is required to use EAP-FAST.

Parameters
- **pac_file** –**[in]** Pointer to the PAC file buffer.
- **pac_file_len** –**[in]** Length of the PAC file buffer.

Returns
- ESP_OK: The PAC file for EAP-FAST authentication was set successfully.

esp_err_t **esp_eap_client_set_fast_params**(*esp_eap_fast_config* config)

Set the parameters for EAP-FAST Phase 1 authentication.

EAP-FAST supports Fast Provisioning, where clients can be authenticated faster using precomputed keys (PAC). This function allows configuring parameters for Fast Provisioning.

Attention 1. Disabling the WPA_MBEDTLS_TLS_CLIENT config is required to use EAP-FAST.

Parameters **config** –**[in]** Configuration structure with Fast Provisioning parameters.

Returns

- ESP_OK: The parameters for EAP-FAST Phase 1 authentication were set successfully.

esp_err_t **esp_eap_client_use_default_cert_bundle**(bool use_default_bundle)

Use the default certificate bundle for EAP authentication.

By default, the EAP client uses a built-in certificate bundle for server verification. Enabling this option allows the use of the default certificate bundle.

Parameters **use_default_bundle** –**[in]** True to use the default certificate bundle, false to use a custom bundle.

Returns

- ESP_OK: The option to use the default certificate bundle was set successfully.

Structures

struct **esp_eap_fast_config**

Configuration settings for EAP-FAST (Extensible Authentication Protocol - Flexible Authentication via Secure Tunneling).

This structure defines the configuration options that can be used to customize the behavior of the EAP-FAST authentication protocol, specifically for Fast Provisioning and PAC (Protected Access Credential) handling.

Public Members

int **fast_provisioning**

Enable or disable Fast Provisioning in EAP-FAST (0 = disabled, 1 = enabled)

int **fast_max_pac_list_len**

Maximum length of the PAC (Protected Access Credential) list

bool **fast_pac_format_binary**

Set to true for binary format PAC, false for ASCII format PAC

Enumerations

enum **esp_eap_ttls_phase2_types**

Enumeration of phase 2 authentication types for EAP-TTLS.

This enumeration defines the supported phase 2 authentication methods that can be used in the EAP-TTLS (Extensible Authentication Protocol - Tunneled Transport Layer Security) protocol for the second authentication phase.

Values:

enumerator **ESP_EAP_TTLS_PHASE2_EAP**

EAP (Extensible Authentication Protocol)

enumerator **ESP_EAP_TTLS_PHASE2_MSCHAPV2**

MS-CHAPv2 (Microsoft Challenge Handshake Authentication Protocol - Version 2)

enumerator **ESP_EAP_TTLS_PHASE2_MSCHAP**

MS-CHAP (Microsoft Challenge Handshake Authentication Protocol)

enumerator **ESP_EAP_TTLS_PHASE2_PAP**

PAP (Password Authentication Protocol)

enumerator **ESP_EAP_TTLS_PHASE2_CHAP**

CHAP (Challenge Handshake Authentication Protocol)

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_wps.h

Functions

esp_err_t **esp_wifi_wps_enable**(const *esp_wps_config_t* *config)

Enable Wi-Fi WPS function.

Parameters **config** –: WPS config to be used in connection

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_FAIL : wps initialization fails

esp_err_t **esp_wifi_wps_disable**(void)

Disable Wi-Fi WPS function and release resource it taken.

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on

esp_err_t **esp_wifi_wps_start**(int timeout_ms)

Start WPS session.

Attention WPS can only be used when station is enabled. WPS needs to be enabled first for using this API.

Parameters **timeout_ms** –: deprecated: This argument' s value will have not effect in functionality of API. The argument will be removed in future. The app should start WPS and register for WIFI events to get the status. WPS status is updated through WPS events. See wifi_event_t enum for more info.

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_ERR_WIFI_WPS_SM : wps state machine is not initialized
- ESP_FAIL : wps initialization fails

esp_err_t **esp_wifi_ap_wps_enable**(const *esp_wps_config_t* *config)

Enable Wi-Fi AP WPS function.

Attention WPS can only be used when softAP is enabled.

Parameters **config** –wps configuration to be used.

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_FAIL : wps initialization fails

esp_err_t **esp_wifi_ap_wps_disable**(void)

Disable Wi-Fi SoftAP WPS function and release resource it taken.

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on

esp_err_t **esp_wifi_ap_wps_start**(const unsigned char *pin)

WPS starts to work.

Attention WPS can only be used when softAP is enabled.

Parameters **pin** –: Pin to be used in case of WPS mode is pin. If Pin is not provided, device will use the pin generated/provided during *esp_wifi_ap_wps_enable()* and reported in WIFI_EVENT_AP_WPS_RG_PIN

Returns

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_ERR_WIFI_WPS_SM : wps state machine is not initialized
- ESP_FAIL : wps initialization fails

Structures

struct **wps_factory_information_t**

Structure representing WPS factory information for ESP device.

This structure holds various strings representing factory information for a device, such as the manufacturer, model number, model name, and device name. Each string is a null-terminated character array. If any of the strings are empty, the default values are used.

Public Members

char **manufacturer**[WPS_MAX_MANUFACTURER_LEN]

Manufacturer of the device. If empty, the default manufacturer is used.

char **model_number**[WPS_MAX_MODEL_NUMBER_LEN]

Model number of the device. If empty, the default model number is used.

char **model_name**[WPS_MAX_MODEL_NAME_LEN]

Model name of the device. If empty, the default model name is used.

char **device_name**[WPS_MAX_DEVICE_NAME_LEN]

Device name. If empty, the default device name is used.

struct **esp_wps_config_t**

Structure representing configuration settings for WPS (Wi-Fi Protected Setup).

This structure encapsulates various configuration settings for WPS, including the WPS type (PBC or PIN), factory information that will be shown in the WPS Information Element (IE), and a PIN if the WPS type is set to PIN.

Public Members

wps_type_t **wps_type**

The type of WPS to be used (PBC or PIN).

wps_factory_information_t **factory_info**

Factory information to be shown in the WPS Information Element (IE). Vendor can choose to display their own information.

char **pin**[PIN_LEN]

WPS PIN (Personal Identification Number) used when wps_type is set to WPS_TYPE_PIN.

Macros

ESP_ERR_WIFI_REGISTRAR

WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE

WPS type error

ESP_ERR_WIFI_WPS_SM

WPS state machine is not initialized

WPS_MAX_MANUFACTURER_LEN

Maximum length of the manufacturer name in WPS information

WPS_MAX_MODEL_NUMBER_LEN

Maximum length of the model number in WPS information

WPS_MAX_MODEL_NAME_LEN

Maximum length of the model name in WPS information

WPS_MAX_DEVICE_NAME_LEN

Maximum length of the device name in WPS information

PIN_LEN

The length of the WPS PIN (Personal Identification Number).

WPS_CONFIG_INIT_DEFAULT(type)

Initialize a default WPS configuration structure with specified WPS type.

This macro initializes a *esp_wps_config_t* structure with default values for the specified WPS type. It sets the WPS type, factory information (including default manufacturer, model number, model name, and device name), and a default PIN value if applicable.

Parameters

- **type** –The WPS type to be used (PBC or PIN).

Returns An initialized *esp_wps_config_t* structure with the specified WPS type and default values.

Type Definitions

typedef enum *wps_type* **wps_type_t**

Enumeration of WPS (Wi-Fi Protected Setup) types.

Enumerations

enum **wps_type**

Enumeration of WPS (Wi-Fi Protected Setup) types.

Values:

enumerator **WPS_TYPE_DISABLE**

WPS is disabled

enumerator **WPS_TYPE_PBC**

WPS Push Button Configuration method

enumerator **WPS_TYPE_PIN**

WPS PIN (Personal Identification Number) method

enumerator **WPS_TYPE_MAX**

Maximum value for WPS type enumeration

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_rrm.h

Functions

int **esp_rrm_send_neighbor_rep_request**(*neighbor_rep_request_cb* cb, void *cb_ctx)

Send Radio measurement neighbor report request to connected AP.

Parameters
- **cb** –callback function for neighbor report
- **cb_ctx** –callback context

Returns
- 0: success
- -1: AP does not support RRM
- -2: station not connected to AP

bool **esp_rrm_is_rrm_supported_connection**(void)

Check RRM capability of connected AP.

Returns
- true: AP supports RRM
- false: AP does not support RRM or station not connected to AP

Type Definitions

typedef void (***neighbor_rep_request_cb**)(void *ctx, const uint8_t *report, size_t report_len)

Callback function type to get neighbor report.

Param ctx neighbor report context
Param report neighbor report
Param report_len neighbor report length
Return
- void

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_wnm.h

Functions

int **esp_wnm_send_bss_transition_mgmt_query**(enum *btm_query_reason* query_reason, const char *btm_candidates, int cand_list)

Send bss transition query to connected AP.

Parameters

- **query_reason** –reason for sending query
- **btm_candidates** –btm candidates list if available
- **cand_list** –whether candidate list to be included from scan results available in supplicant's cache.

Returns

- 0: success
- -1: AP does not support BTM
- -2: station not connected to AP

bool **esp_wnm_is_btm_supported_connection**(void)

Check bss trasition capability of connected AP.

Returns

- true: AP supports BTM
- false: AP does not support BTM or station not connected to AP

Enumerations

enum **btm_query_reason**

enum btm_query_reason: Reason code for sending btm query

Values:

enumerator **REASON_UNSPECIFIED**

enumerator **REASON_FRAME_LOSS**

enumerator **REASON_DELAY**

enumerator **REASON_BANDWIDTH**

enumerator **REASON_LOAD_BALANCE**

enumerator **REASON_RSSI**

enumerator **REASON_RETRANSMISSIONS**

enumerator **REASON_INTERFERENCE**

enumerator **REASON_GRAY_ZONE**

enumerator **REASON_PREMIUM_AP**

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_mbo.h

Functions

int **esp_mbo_update_non_pref_chan**(struct *non_pref_chan_s* *non_pref_chan)

Update channel preference for MBO IE.

Parameters **non_pref_chan** –Non preference channel list

Returns

- 0: success else failure

Structures

struct **non_pref_chan**

Structure representing a non-preferred channel in a wireless network.

This structure encapsulates information about a non-preferred channel including the reason for its non-preference, the operating class, channel number, and preference level.

Public Members

enum *non_pref_chan_reason* **reason**

Reason for the channel being non-preferred

uint8_t **oper_class**

Operating class of the channel

uint8_t **chan**

Channel number

uint8_t **preference**

Preference level of the channel

struct **non_pref_chan_s**

Structure representing a list of non-preferred channels in a wireless network.

This structure encapsulates information about a list of non-preferred channels including the number of non-preferred channels and an array of structures representing individual non-preferred channels.

Public Members

size_t **non_pref_chan_num**

Number of non-preferred channels in the list

struct *non_pref_chan* **chan**[]

Array of structures representing individual non-preferred channels

Enumerations

enum **non_pref_chan_reason**

Enumeration of reasons for a channel being non-preferred in a wireless network.

This enumeration defines various reasons why a specific channel might be considered non-preferred in a wireless network configuration.

Values:

enumerator **NON_PREF_CHAN_REASON_UNSPECIFIED**

Unspecified reason for non-preference

enumerator **NON_PREF_CHAN_REASON_RSSI**

Non-preferred due to low RSSI (Received Signal Strength Indication)

enumerator **NON_PREF_CHAN_REASON_EXT_INTERFERENCE**

Non-preferred due to external interference

enumerator **NON_PREF_CHAN_REASON_INT_INTERFERENCE**

Non-preferred due to internal interference

Wi-Fi Easy Connect™ (DPP)

Wi-Fi Easy Connect™, also known as Device Provisioning Protocol (DPP) or Easy Connect, is a provisioning protocol certified by Wi-Fi Alliance. It is a secure and standardized provisioning protocol for configuration of Wi-Fi Devices. With Easy Connect adding a new device to a network is as simple as scanning a QR Code. This reduces complexity and enhances user experience while onboarding devices without UI like Smart Home and IoT products. Unlike old protocols like WiFi Protected Setup (WPS), Wi-Fi Easy Connect incorporates strong encryption through public key cryptography to ensure networks remain secure as new devices are added. Easy Connect brings many benefits in the User Experience:

- Simple and intuitive to use; no lengthy instructions to follow for new device setup
- No need to remember and enter passwords into the device being provisioned
- Works with electronic or printed QR codes, or human-readable strings
- Supports both WPA2 and WPA3 networks

Please refer to Wi-Fi Alliance' s official page on Easy Connect for more information.

ESP32 supports Enrollee mode of Easy Connect with QR Code as the provisioning method. A display is required to display this QR Code. Users can scan this QR Code using their capable device and provision the ESP32 to their Wi-Fi network. The provisioning device needs to be connected to the AP which need not support Wi-Fi Easy Connect™. Easy Connect is still an evolving protocol. Of known platforms that support the QR Code method are some Android smartphones with Android 10 or higher. To use Easy Connect no additional App needs to be installed on the supported smartphone.

Application Example Example on how to provision ESP32 using a supported smartphone: wifi/wifi_easy_connect/dpp-enrollee.

API Reference

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_dpp.h

Functions

esp_err_t **esp_supp_dpp_init** (*esp_supp_dpp_event_cb_t* evt_cb)

Initialize DPP Supplicant.

```
Starts DPP Supplicant and initializes related Data Structures.
```

return

- ESP_OK: Success
- ESP_FAIL: Failure

Parameters **evt_cb** –Callback function to receive DPP related events

void **esp_supp_dpp_deinit**(void)

De-initalize DPP Supplicant.

```
Frees memory from DPP Supplicant Data Structures.
```

esp_err_t **esp_supp_dpp_bootstrap_gen**(const char *chan_list, *esp_supp_dpp_bootstrap_t* type, const char *key, const char *info)

Generates Bootstrap Information as an Enrollee.

```
Generates Out Of Band Bootstrap information as an Enrollee which can be
used by a DPP Configurator to provision the Enrollee.
```

Parameters

- **chan_list** –List of channels device will be available on for listening
- **type** –Bootstrap method type, only QR Code method is supported for now.
- **key** –(Optional) 32 byte Raw Private Key for generating a Bootstrapping Public Key
- **info** –(Optional) Ancilliary Device Information like Serial Number

Returns

- ESP_OK: Success
- ESP_FAIL: Failure

esp_err_t **esp_supp_dpp_start_listen**(void)

Start listening on Channels provided during esp_supp_dpp_bootstrap_gen.

```
Listens on every Channel from Channel List for a pre-defined wait time.
```

Returns

- ESP_OK: Success
- ESP_FAIL: Generic Failure
- ESP_ERR_INVALID_STATE: ROC attempted before WiFi is started
- ESP_ERR_NO_MEM: Memory allocation failed while posting ROC request

void **esp_supp_dpp_stop_listen**(void)

Stop listening on Channels.

```
Stops listening on Channels and cancels ongoing listen operation.
```

Macros

ESP_DPP_AUTH_TIMEOUT_SECS

ESP_ERR_DPP_FAILURE

Generic failure during DPP Operation

ESP_ERR_DPP_TX_FAILURE

DPP Frame Tx failed OR not Acked

ESP_ERR_DPP_INVALID_ATTR

Encountered invalid DPP Attribute

ESP_ERR_DPP_AUTH_TIMEOUT

DPP Auth response was not recieved in time

Type Definitions

typedef enum *dpp_bootstrap_type* **esp_supp_dpp_bootstrap_t**

Types of Bootstrap Methods for DPP.

typedef void (***esp_supp_dpp_event_cb_t**)(*esp_supp_dpp_event_t* evt, void *data)

Callback function for receiving DPP Events from Supplicant.

```
Callback function will be called with DPP related information.
```

Param evt DPP event ID
Param data Event data payload

Enumerations

enum **dpp_bootstrap_type**

Types of Bootstrap Methods for DPP.

Values:

enumerator **DPP_BOOTSTRAP_QR_CODE**

QR Code Method

enumerator **DPP_BOOTSTRAP_PKEX**

Proof of Knowledge Method

enumerator **DPP_BOOTSTRAP_NFC_URI**

NFC URI record Method

enum **esp_supp_dpp_event_t**

Types of Callback Events received from DPP Supplicant.

Values:

enumerator **ESP_SUPP_DPP_URI_READY**

URI is ready through Bootstrapping

enumerator **ESP_SUPP_DPP_CFG_RECVD**

Config received via DPP Authentication

enumerator **ESP_SUPP_DPP_FAIL**

DPP Authentication failure

Code examples for the Wi-Fi API are provided in the wifi directory of ESP-IDF examples.

Code examples for ESP-WIFI-MESH are provided in the mesh directory of ESP-IDF examples.

2.5.2 Ethernet

Ethernet

Overview ESP-IDF provides a set of consistent and flexible APIs to support both internal Ethernet MAC (EMAC) controller and external SPI-Ethernet modules.

This programming guide is split into the following sections:

1. *Basic Ethernet Concepts*
2. *Configure MAC and PHY*
3. *Connect Driver to TCP/IP Stack*
4. *Misc control of Ethernet driver*

Basic Ethernet Concepts Ethernet is an asynchronous Carrier Sense Multiple Access with Collision Detect (CSMA/CD) protocol/interface. It is generally not well suited for low power applications. However, with ubiquitous deployment, internet connectivity, high data rates and limitless rage expandability, Ethernet can accommodate nearly all wired communications.

Normal IEEE 802.3 compliant Ethernet frames are between 64 and 1518 bytes in length. They are made up of five or six different fields: a destination MAC address (DA), a source MAC address (SA), a type/length field, data payload, an optional padding field and a Cyclic Redundancy Check (CRC). Additionally, when transmitted on the Ethernet medium, a 7-byte preamble field and Start-of-Frame (SOF) delimiter byte are appended to the beginning of the Ethernet packet.

Thus the traffic on the twist-pair cabling will appear as shown blow:

Preamble and Start-of-Frame Delimiter The preamble contains seven bytes of `55H`, it allows the receiver to lock onto the stream of data before the actual frame arrives. The Start-of-Frame Delimiter (SFD) is a binary sequence `10101011` (as seen on the physical medium). It is sometimes considered to be part of the preamble.

When transmitting and receiving data, the preamble and SFD bytes will automatically be generated or stripped from the packets.

Destination Address The destination address field contains a 6-byte length MAC address of the device that the packet is directed to. If the Least Significant bit in the first byte of the MAC address is set, the address is a multi-cast destination. For example, 01-00-00-00-F0-00 and 33-45-67-89-AB-CD are multi-cast addresses, while 00-00-00-00-F0-00 and 32-45-67-89-AB-CD are not. Packets with multi-cast destination addresses are designed to arrive and be important to a selected group of Ethernet nodes. If the destination address field is the reserved multi-cast address, i.e. FF-FF-FF-FF-FF-FF, the packet is a broadcast packet and it will be directed to everyone sharing the network. If the Least Significant bit in the first byte of the MAC address is clear, the address is a uni-cast address and will be designed for usage by only the addressed node.

Normally the EMAC controller incorporates receive filters which can be used to discard or accept packets with multi-cast, broadcast and/or uni-cast destination addresses. When transmitting packets, the host controller is responsible for writing the desired destination address into the transmit buffer.

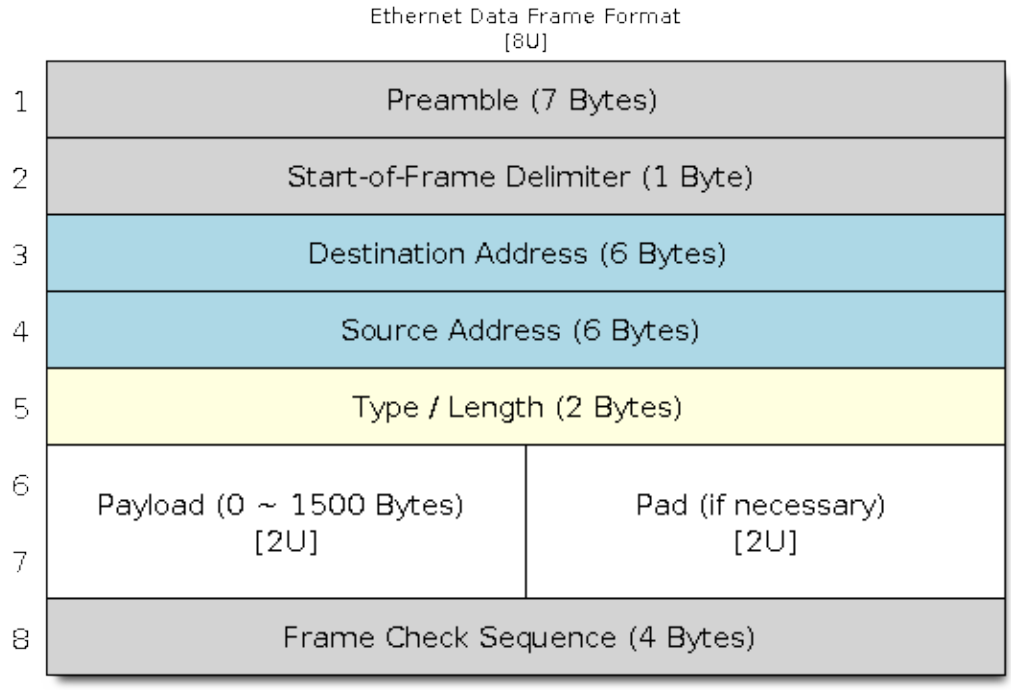


Fig. 4: Ethernet Data Frame Format

Source Address The source address field contains a 6-byte length MAC address of the node which created the Ethernet packet. Users of Ethernet must generate a unique MAC address for each controller used. MAC addresses consist of two portions. The first three bytes are known as the Organizationally Unique Identifier (OUI). OUIs are distributed by the IEEE. The last three bytes are address bytes at the discretion of the company that purchased the OUI. More information about MAC Address used in ESP-IDF, please see *MAC Address Allocation*.

When transmitting packets, the assigned source MAC address must be written into the transmit buffer by the host controller.

Type / Length The type/length field is a 2-byte field, if the value in this field is <= 1500 (decimal), it is considered a length field and it specifies the amount of non-padding data which follows in the data field. If the value is >= 1536, it represents the protocol the following packet data belongs to. The following are the most common type values:

- IPv4 = 0800H
- IPv6 = 86DDH
- ARP = 0806H

Users implementing proprietary networks may choose to treat this field as a length field, while applications implementing protocols such as the Internet Protocol (IP) or Address Resolution Protocol (ARP), should program this field with the appropriate type defined by the protocol' s specification when transmitting packets.

Payload The payload field is a variable length field, anywhere from 0 to 1500 bytes. Larger data packets will violate Ethernet standards and will be dropped by most Ethernet nodes. This field contains the client data, such as an IP datagram.

Padding and FCS The padding field is a variable length field added to meet IEEE 802.3 specification requirements when small data payloads are used. The DA, SA, type, payload and padding of an Ethernet packet must be no smaller

than 60 bytes. Adding the required 4-byte FCS field, packets must be no smaller than 64 bytes. If the data field is less than 46 bytes long, a padding field is required.

The FCS field is a 4-byte field which contains an industry standard 32-bit CRC calculated with the data from the DA, SA, type, payload and padding fields. Given the complexity of calculating a CRC, the hardware normally will automatically generate a valid CRC and transmit it. Otherwise, the host controller must generate the CRC and place it in the transmit buffer.

Normally, the host controller does not need to concern itself with padding and the CRC which the hardware EMAC will also be able to automatically generate when transmitting and verify when receiving. However, the padding and CRC fields will be written into the receive buffer when packets arrive, so they may be evaluated by the host controller if needed.

Note: Besides the basic data frame described above, there're two other common frame types in 10/100 Mbps Ethernet: control frames and VLAN tagged frames. They're not supported in ESP-IDF.

Configure MAC and PHY Ethernet driver is composed of two parts: MAC and PHY.

The communication between MAC and PHY can have diverse choices: **MII** (Media Independent Interface), **RMII** (Reduced Media Independent Interface) and etc.

One of the obvious difference between MII and RMII is the signal consumption. For MII, it usually costs up to 18 signals. Instead, RMII interface can reduce the consumption to 9.

In RMII mode, both the receiver and transmitter signals are referenced to the `REF_CLK`. **REF_CLK must be stable during any access to PHY and MAC**. Generally there're three ways to generate the `REF_CLK` depending on the PHY device in your design:

- Some PHY chip can derive the `REF_CLK` from its external connected 25MHz crystal oscillator (as seen the option *a* in the picture). In this case, you should select `CONFIG_ETH_RMII_CLK_INPUT` in *CONFIG_ETH_RMII_CLK_MODE*.
- Some PHY chip uses an external connected 50MHz crystal oscillator or other clock source, which can also be used as the `REF_CLK` for MAC side (as seen the option *b* in the picture). In this case, you still need to select `CONFIG_ETH_RMII_CLK_INPUT` in *CONFIG_ETH_RMII_CLK_MODE*.
- Some EMAC controller can generate the `REF_CLK` using its internal high precision PLL (as seen the option *c* in the picture). In this case, you should select `CONFIG_ETH_RMII_CLK_OUTPUT` in *CONFIG_ETH_RMII_CLK_MODE*.

Note: `REF_CLK` is configured via Project Configuration as described above by default. However, it can be overwritten from user application code by appropriately setting *`eth_esp32_emac_config_t::interface`* and *`eth_esp32_emac_config_t::clock_config`* members. See *`emac_rmii_clock_mode_t`* and *`emac_rmii_clock_gpio_t`* for more details.

Warning: If the RMII clock mode is selected to `CONFIG_ETH_RMII_CLK_OUTPUT`, then `GPIO0` can be used to output the `REF_CLK` signal. See *CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0* for more information. What's more, if you're not using PSRAM in your design, GPIO16 and GPIO17 are also available to output the reference clock. See *CONFIG_ETH_RMII_CLK_OUT_GPIO* for more information.

If the RMII clock mode is selected to `CONFIG_ETH_RMII_CLK_INPUT`, then `GPIO0` is the only choice to input the `REF_CLK` signal. Please note that, `GPIO0` is also an important strapping GPIO on ESP32. If GPIO0 samples a low level during power up, ESP32 will go into download mode. The system will get halted until a manually reset. The workaround of this issue is disabling the `REF_CLK` in hardware by default, so that the strapping pin won't be interfered by other signals in boot stage. Then re-enable the `REF_CLK` in Ethernet driver installation stage. The ways to disable the `REF_CLK` signal can be:

- Disable or power down the crystal oscillator (as the case *b* in the picture).

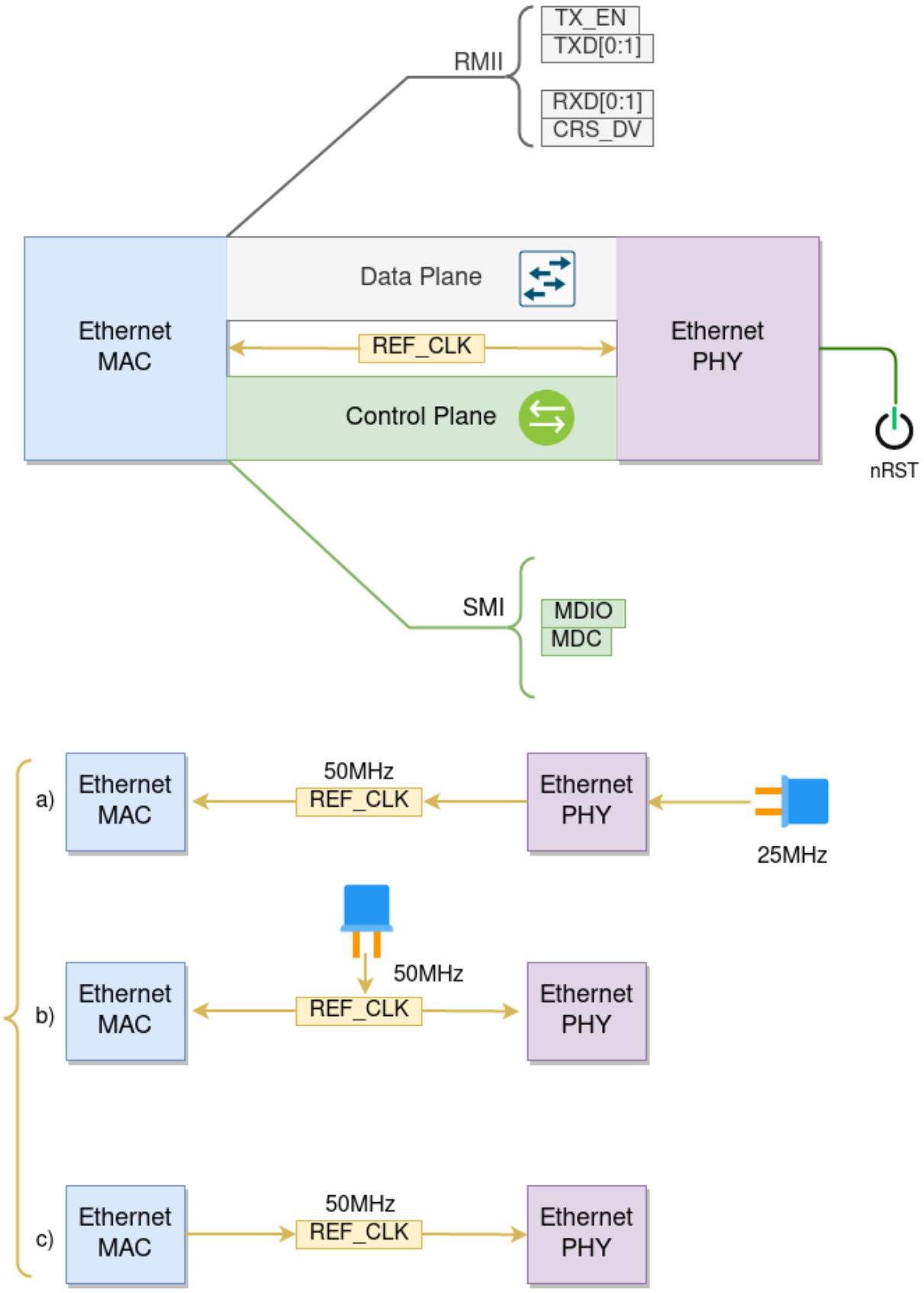


Fig. 5: Ethernet RMII Interface

- Force the PHY device in reset status (as the case *a* in the picture). **This could fail for some PHY device** (i.e. it still outputs signal to GPIO0 even in reset state).

No matter which RMII clock mode you select, you really need to take care of the signal integrity of REF_CLK in your hardware design! Keep the trace as short as possible. Keep it away from RF devices. Keep it away from inductor elements.

Note: ESP-IDF only supports the RMII interface (i.e. always select `CONFIG_ETH_PHY_INTERFACE_RMII` in Kconfig option *CONFIG_ETH_PHY_INTERFACE*).

Signals used in data plane are fixed to specific GPIOs via MUX, they can't be modified to other GPIOs. Signals used in control plane can be routed to any free GPIOs via Matrix. Please refer to *ESP32-Ethernet-Kit* for hardware design example.

We need to setup necessary parameters for MAC and PHY respectively based on your Ethernet board design and then combine the two together, completing the driver installation.

Configuration for MAC is described in *eth_mac_config_t*, including:

- *eth_mac_config_t::sw_reset_timeout_ms*: software reset timeout value, in milliseconds, typically MAC reset should be finished within 100ms.
- *eth_mac_config_t::rx_task_stack_size* and *eth_mac_config_t::rx_task_prio*: the MAC driver creates a dedicated task to process incoming packets, these two parameters are used to set the stack size and priority of the task.
- *eth_mac_config_t::flags*: specifying extra features that the MAC driver should have, it could be useful in some special situations. The value of this field can be OR'd with macros prefixed with `ETH_MAC_FLAG_`. For example, if the MAC driver should work when cache is disabled, then you should configure this field with *ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE*.
- *eth_esp32_emac_config_t::smi_mdc_gpio_num* and *eth_esp32_emac_config_t::smi_mdio_gpio_n* the GPIO number used to connect the SMI signals.
- *eth_esp32_emac_config_t::interface*: configuration of MAC Data interface to PHY (MII/RMII).
- *eth_esp32_emac_config_t::clock_config*: configuration of EMAC Interface clock (`REF_CLK` mode and GPIO number in case of RMII).

Configuration for PHY is described in *eth_phy_config_t*, including:

- *eth_phy_config_t::phy_addr*: multiple PHY device can share the same SMI bus, so each PHY needs a unique address. Usually this address is configured during hardware design by pulling up/down some PHY strapping pins. You can set the value from 0 to 15 based on your Ethernet board. Especially, if the SMI bus is shared by only one PHY device, setting this value to -1 can enable the driver to detect the PHY address automatically.
- *eth_phy_config_t::reset_timeout_ms*: reset timeout value, in milliseconds, typically PHY reset should be finished within 100ms.
- *eth_phy_config_t::autonego_timeout_ms*: auto-negotiation timeout value, in milliseconds. Ethernet driver will start negotiation with the peer Ethernet node automatically, to determine to duplex and speed mode. This value usually depends on the ability of the PHY device on your board.
- *eth_phy_config_t::reset_gpio_num*: if your board also connect the PHY reset pin to one of the GPIO, then set it here. Otherwise, set this field to -1.

ESP-IDF provides a default configuration for MAC and PHY in macro *ETH_MAC_DEFAULT_CONFIG* and *ETH_PHY_DEFAULT_CONFIG*.

Create MAC and PHY Instance Ethernet driver is implemented in an Object-Oriented style. Any operation on MAC and PHY should be based on the instance of them two.

Internal EMAC + External PHY

```
eth_mac_config_t mac_config = ETH_MAC_DEFAULT_CONFIG();                          //␣
↪apply default common MAC configuration
eth_esp32_emac_config_t esp32_emac_config = ETH_ESP32_EMAC_DEFAULT_CONFIG(); //␣
↪apply default vendor-specific MAC configuration
esp32_emac_config.smi_mdc_gpio_num = CONFIG_EXAMPLE_ETH_MDC_GPIO;            //␣
↪alter the GPIO used for MDC signal
esp32_emac_config.smi_mdio_gpio_num = CONFIG_EXAMPLE_ETH_MDIO_GPIO;          //␣
↪alter the GPIO used for MDIO signal
esp_eth_mac_t *mac = esp_eth_mac_new_esp32(&esp32_emac_config, &mac_config); //␣
↪create MAC instance

eth_phy_config_t phy_config = ETH_PHY_DEFAULT_CONFIG();      // apply default PHY␣
↪configuration
phy_config.phy_addr = CONFIG_EXAMPLE_ETH_PHY_ADDR;           // alter the PHY␣
↪address according to your board design
phy_config.reset_gpio_num = CONFIG_EXAMPLE_ETH_PHY_RST_GPIO; // alter the GPIO␣
↪used for PHY reset
esp_eth_phy_t *phy = esp_eth_phy_new_ip101(&phy_config);     // create PHY instance
// ESP-IDF officially supports several different Ethernet PHY chip driver
// esp_eth_phy_t *phy = esp_eth_phy_new_rtl8201(&phy_config);
// esp_eth_phy_t *phy = esp_eth_phy_new_lan8720(&phy_config);
// esp_eth_phy_t *phy = esp_eth_phy_new_dp83848(&phy_config);
```

Optional Runtime MAC Clock Configuration EMAC `REF_CLK` can be optionally configured from user application code.

```
eth_esp32_emac_config_t esp32_emac_config = ETH_ESP32_EMAC_DEFAULT_CONFIG(); //␣
↪apply default vendor-specific MAC configuration

// ...

esp32_emac_config.interface = EMAC_DATA_INTERFACE_RMII;                          //␣
↪alter EMAC Data Interface
esp32_emac_config.clock_config.rmii.clock_mode = EMAC_CLK_OUT;                   //␣
↪select EMAC REF_CLK mode
esp32_emac_config.clock_config.rmii.clock_gpio = EMAC_CLK_OUT_GPIO;              //␣
↪select GPIO number used to input/output EMAC REF_CLK
esp_eth_mac_t *mac = esp_eth_mac_new_esp32(&esp32_emac_config, &mac_config); //␣
↪create MAC instance
```

SPI-Ethernet Module

```
eth_mac_config_t mac_config = ETH_MAC_DEFAULT_CONFIG();      // apply default␣
↪common MAC configuration
eth_phy_config_t phy_config = ETH_PHY_DEFAULT_CONFIG();      // apply default PHY␣
↪configuration
phy_config.phy_addr = CONFIG_EXAMPLE_ETH_PHY_ADDR;           // alter the PHY␣
↪address according to your board design
phy_config.reset_gpio_num = CONFIG_EXAMPLE_ETH_PHY_RST_GPIO; // alter the GPIO␣
↪used for PHY reset
// Install GPIO interrupt service (as the SPI-Ethernet module is interrupt driven)
gpio_install_isr_service(0);
// SPI bus configuration
spi_device_handle_t spi_handle = NULL;
spi_bus_config_t buscfg = {
    .miso_io_num = CONFIG_EXAMPLE_ETH_SPI_MISO_GPIO,
    .mosi_io_num = CONFIG_EXAMPLE_ETH_SPI_MOSI_GPIO,
    .sclk_io_num = CONFIG_EXAMPLE_ETH_SPI_SCLK_GPIO,
    .quadwp_io_num = -1,
```

(continues on next page)

(continued from previous page)

```
    .quadhd_io_num = -1,
};
ESP_ERROR_CHECK(spi_bus_initialize(CONFIG_EXAMPLE_ETH_SPI_HOST, &buscfg, 1));
// Configure SPI device
spi_device_interface_config_t spi_devcfg = {
    .mode = 0,
    .clock_speed_hz = CONFIG_EXAMPLE_ETH_SPI_CLOCK_MHZ * 1000 * 1000,
    .spics_io_num = CONFIG_EXAMPLE_ETH_SPI_CS_GPIO,
    .queue_size = 20
};
/* dm9051 ethernet driver is based on spi driver */
eth_dm9051_config_t dm9051_config = ETH_DM9051_DEFAULT_CONFIG(CONFIG_EXAMPLE_ETH_
↪SPI_HOST, &spi_devcfg);
dm9051_config.int_gpio_num = CONFIG_EXAMPLE_ETH_SPI_INT_GPIO;
esp_eth_mac_t *mac = esp_eth_mac_new_dm9051(&dm9051_config, &mac_config);
esp_eth_phy_t *phy = esp_eth_phy_new_dm9051(&phy_config);
```

Note:

- When creating MAC and PHY instance for SPI-Ethernet modules (e.g. DM9051), the constructor function must have the same suffix (e.g. *esp_eth_mac_new_dm9051* and *esp_eth_phy_new_dm9051*). This is because we don't have other choices but the integrated PHY.
- The SPI device configuration (i.e. *spi_device_interface_config_t*) may slightly differ for other Ethernet modules or to meet SPI timing on specific PCB. Please check out your module's spec and the examples in esp-idf.

Install Driver To install the Ethernet driver, we need to combine the instance of MAC and PHY and set some additional high-level configurations (i.e. not specific to either MAC or PHY) in *esp_eth_config_t*:

- *esp_eth_config_t::mac*: instance that created from MAC generator (e.g. *esp_eth_mac_new_esp32()*).
- *esp_eth_config_t::phy*: instance that created from PHY generator (e.g. *esp_eth_phy_new_ip101()*).
- *esp_eth_config_t::check_link_period_ms*: Ethernet driver starts an OS timer to check the link status periodically, this field is used to set the interval, in milliseconds.
- *esp_eth_config_t::stack_input*: In most of Ethernet IoT applications, any Ethernet frame that received by driver should be passed to upper layer (e.g. TCP/IP stack). This field is set to a function which is responsible to deal with the incoming frames. You can even update this field at runtime via function *esp_eth_update_input_path()* after driver installation.
- *esp_eth_config_t::on_lowlevel_init_done* and *esp_eth_config_t::on_lowlevel_deinit_done*: These two fields are used to specify the hooks which get invoked when low level hardware has been initialized or de-initialized.

ESP-IDF provides a default configuration for driver installation in macro *ETH_DEFAULT_CONFIG*.

```
esp_eth_config_t config = ETH_DEFAULT_CONFIG(mac, phy); // apply default driver␣
↪configuration
esp_eth_handle_t eth_handle = NULL; // after driver installed, we will get the␣
↪handle of the driver
esp_eth_driver_install(&config, &eth_handle); // install driver
```

Ethernet driver also includes event-driven model, which will send useful and important event to user space. We need to initialize the event loop before installing the Ethernet driver. For more information about event-driven programming, please refer to *ESP Event*.

```
/** Event handler for Ethernet events */
static void eth_event_handler(void *arg, esp_event_base_t event_base,
                              int32_t event_id, void *event_data)
```

(continues on next page)

(continued from previous page)

```
{
    uint8_t mac_addr[6] = {0};
    /* we can get the ethernet driver handle from event data */
    esp_eth_handle_t eth_handle = *(esp_eth_handle_t *)event_data;

    switch (event_id) {
    case ETHERNET_EVENT_CONNECTED:
        esp_eth_ioctl(eth_handle, ETH_CMD_G_MAC_ADDR, mac_addr);
        ESP_LOGI(TAG, "Ethernet Link Up");
        ESP_LOGI(TAG, "Ethernet HW Addr %02x:%02x:%02x:%02x:%02x:%02x",
                 mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_
↪addr[4], mac_addr[5]);
        break;
    case ETHERNET_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "Ethernet Link Down");
        break;
    case ETHERNET_EVENT_START:
        ESP_LOGI(TAG, "Ethernet Started");
        break;
    case ETHERNET_EVENT_STOP:
        ESP_LOGI(TAG, "Ethernet Stopped");
        break;
    default:
        break;
    }
}

esp_event_loop_create_default(); // create a default event loop that running in␣
↪background
esp_event_handler_register(ETH_EVENT, ESP_EVENT_ANY_ID, &eth_event_handler, NULL);␣
↪// register Ethernet event handler (to deal with user specific stuffs when event␣
↪like link up/down happened)
```

Start Ethernet Driver After driver installation, we can start Ethernet immediately.

```
esp_eth_start(eth_handle); // start Ethernet driver state machine
```

Connect Driver to TCP/IP Stack Up until now, we have installed the Ethernet driver. From the view of OSI (Open System Interconnection), we're still on level 2 (i.e. Data Link Layer). We can detect link up and down event, we can gain MAC address in user space, but we can't obtain IP address, let alone send HTTP request. The TCP/IP stack used in ESP-IDF is called LwIP, for more information about it, please refer to *LwIP*.

To connect Ethernet driver to TCP/IP stack, these three steps need to follow:

1. Create network interface for Ethernet driver
2. Attach the network interface to Ethernet driver
3. Register IP event handlers

More information about network interface, please refer to *Network Interface*.

```
/** Event handler for IP_EVENT_ETH_GOT_IP */
static void got_ip_event_handler(void *arg, esp_event_base_t event_base,
                                 int32_t event_id, void *event_data)
{
    ip_event_got_ip_t *event = (ip_event_got_ip_t *) event_data;
    const esp_netif_ip_info_t *ip_info = &event->ip_info;

    ESP_LOGI(TAG, "Ethernet Got IP Address");
    ESP_LOGI(TAG, "~~~~~~~~~~~");
```

(continues on next page)

(continued from previous page)

```
    ESP_LOGI(TAG, "ETHIP:" IPSTR, IP2STR(&ip_info->ip));
    ESP_LOGI(TAG, "ETHMASK:" IPSTR, IP2STR(&ip_info->netmask));
    ESP_LOGI(TAG, "ETHGW:" IPSTR, IP2STR(&ip_info->gw));
    ESP_LOGI(TAG, "~~~~~~~~~~~");
}

esp_netif_init()); // Initialize TCP/IP network interface (should be called only␣
↪once in application)
esp_netif_config_t cfg = ESP_NETIF_DEFAULT_ETH(); // apply default network␣
↪interface configuration for Ethernet
esp_netif_t *eth_netif = esp_netif_new(&cfg); // create network interface for␣
↪Ethernet driver

esp_netif_attach(eth_netif, esp_eth_new_netif_glue(eth_handle)); // attach␣
↪Ethernet driver to TCP/IP stack
esp_event_handler_register(IP_EVENT, IP_EVENT_ETH_GOT_IP, &got_ip_event_handler,␣
↪NULL); // register user defined IP event handlers
esp_eth_start(eth_handle); // start Ethernet driver state machine
```

Warning: It is recommended to fully initialize the Ethernet driver and network interface prior registering user's Ethernet/IP event handlers, i.e. register the event handlers as the last thing prior starting the Ethernet driver. Such approach ensures that Ethernet/IP events get executed first by the Ethernet driver or network interface and so the system is in expected state when executing user's handlers.

Misc control of Ethernet driver The following functions should only be invoked after the Ethernet driver has been installed.

- Stop Ethernet driver: *esp_eth_stop()*
- Update Ethernet data input path: *esp_eth_update_input_path()*
- Misc get/set of Ethernet driver attributes: *esp_eth_ioctl()*

```
/* get MAC address */
uint8_t mac_addr[6];
memset(mac_addr, 0, sizeof(mac_addr));
esp_eth_ioctl(eth_handle, ETH_CMD_G_MAC_ADDR, mac_addr);
ESP_LOGI(TAG, "Ethernet MAC Address: %02x:%02x:%02x:%02x:%02x:%02x",
         mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4], mac_
↪addr[5]);

/* get PHY address */
int phy_addr = -1;
esp_eth_ioctl(eth_handle, ETH_CMD_G_PHY_ADDR, &phy_addr);
ESP_LOGI(TAG, "Ethernet PHY Address: %d", phy_addr);
```

Flow control Ethernet on MCU usually has a limitation in the number of frames it can handle during network congestion, because of the limitation in RAM size. A sending station might be transmitting data faster than the peer end can accept it. Ethernet flow control mechanism allows the receiving node to signal the sender requesting suspension of transmissions until the receiver catches up. The magic behind that is the pause frame, which was defined in IEEE 802.3x.

Pause frame is a special Ethernet frame used to carry the pause command, whose EtherType field is 0x8808, with the Control opcode set to 0x0001. Only stations configured for full-duplex operation may send pause frames. When a station wishes to pause the other end of a link, it sends a pause frame to the 48-bit reserved multicast address of 01-80-C2-00-00-01. The pause frame also includes the period of pause time being requested, in the form of a two-byte integer, ranging from 0 to 65535.

After Ethernet driver installation, the flow control feature is disabled by default. You can enable it by:

```
bool flow_ctrl_enable = true;
esp_eth_ioctl(eth_handle, ETH_CMD_S_FLOW_CTRL, &flow_ctrl_enable);
```

One thing should be kept in mind, is that the pause frame ability will be advertised to peer end by PHY during auto negotiation. Ethernet driver sends pause frame only when both sides of the link support it.

Application Examples

- Ethernet basic example: ethernet/basic.
- Ethernet iperf example: ethernet/iperf.
- Ethernet to Wi-Fi AP "router" : ethernet/eth2ap.
- Most of protocol examples should also work for Ethernet: protocols.

Advanced Topics

Custom PHY Driver There are multiple PHY manufactures with their wide portfolios of chips available. The ESP-IDF already supports several PHY chips however one can easily get to a point where none of them satisfies user's actual needs due to either price, features, stock availability etc.

Luckily, a management interface between EMAC and PHY is standardized by IEEE 802.3 in 22.2.4 Management functions section. It defines provisions of so called "MII Management Interface" for the purposes of controlling the PHY and gathering status from the PHY. A set of management registers is defined to control chip behavior, link properties, auto-negotiation configuration etc. This basic management functionality is addressed by esp_eth/src/esp_eth_phy_802_3.c in ESP-IDF and so it makes a creation of new custom PHY chip driver quite a simple task.

Note: Always consult with PHY datasheet since some PHY chips may not comply with IEEE 802.3, Section 22.2.4. It does not mean you are not able to create a custom PHY driver, it will just require more effort. You will have to define all PHY management functions.

Majority of PHY management functionality required by the ESP-IDF Ethernet driver is covered by the esp_eth/src/esp_eth_phy_802_3.c however, the following may require developing chip specific management functions:

- link status which is almost always chip specific,
- chip initialization, even though it is not strictly required, should be customized to at least ensure that expected chip is used and
- chip specific features configuration.

Steps to create custom PHY driver:

1. Define vendor specific registry layout based on PHY datasheet. See esp_eth/src/esp_eth_phy_ip101.c as an example.
2. Prepare derived PHY management object infostructure which
   - must contain at least parent IEEE 802.3 *phy_802_3_t* object and
   - optionally contain additional variables needed to support non-IEEE 802.3 or customized functionality. See esp_eth/src/esp_eth_phy_ksz80xx.c as an example.
3. Define chip specific management call-back functions.
4. Initialize parent IEEE 802.3 object and re-assign chip specific management call-back functions.

Once you finish the new custom PHY driver implementation, consider sharing it among with other users via IDF Component Registry.

API Reference

Header File

- components/esp_eth/include/esp_eth.h

Header File

- components/esp_eth/include/esp_eth_driver.h

Functions

esp_err_t **esp_eth_driver_install** (const *esp_eth_config_t* *config, *esp_eth_handle_t* *out_hdl)

Install Ethernet driver.

Parameters
- **config** –**[in]** configuration of the Ethernet driver
- **out_hdl** –**[out]** handle of Ethernet driver

Returns
- ESP_OK: install esp_eth driver successfully
- ESP_ERR_INVALID_ARG: install esp_eth driver failed because of some invalid argument
- ESP_ERR_NO_MEM: install esp_eth driver failed because there's no memory for driver
- ESP_FAIL: install esp_eth driver failed because some other error occurred

esp_err_t **esp_eth_driver_uninstall** (*esp_eth_handle_t* hdl)

Uninstall Ethernet driver.

Note: It's not recommended to uninstall Ethernet driver unless it won't get used any more in application code. To uninstall Ethernet driver, you have to make sure, all references to the driver are released. Ethernet driver can only be uninstalled successfully when reference counter equals to one.

Parameters **hdl** –**[in]** handle of Ethernet driver

Returns
- ESP_OK: uninstall esp_eth driver successfully
- ESP_ERR_INVALID_ARG: uninstall esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: uninstall esp_eth driver failed because it has more than one reference
- ESP_FAIL: uninstall esp_eth driver failed because some other error occurred

esp_err_t **esp_eth_start** (*esp_eth_handle_t* hdl)

Start Ethernet driver **ONLY** in standalone mode (i.e. without TCP/IP stack)

Note: This API will start driver state machine and internal software timer (for checking link status).

Parameters **hdl** –**[in]** handle of Ethernet driver

Returns
- ESP_OK: start esp_eth driver successfully
- ESP_ERR_INVALID_ARG: start esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: start esp_eth driver failed because driver has started already
- ESP_FAIL: start esp_eth driver failed because some other error occurred

esp_err_t **esp_eth_stop** (*esp_eth_handle_t* hdl)

Stop Ethernet driver.

Note: This function does the oppsite operation of `esp_eth_start`.

Parameters **`hdl`** –**[in]** handle of Ethernet driver

Returns

- ESP_OK: stop esp_eth driver successfully
- ESP_ERR_INVALID_ARG: stop esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: stop esp_eth driver failed because driver has not started yet
- ESP_FAIL: stop esp_eth driver failed because some other error occurred

esp_err_t **`esp_eth_update_input_path`** (*esp_eth_handle_t* hdl, *esp_err_t* (*stack_input)(*esp_eth_handle_t* hdl, uint8_t *buffer, uint32_t length, void *priv), void *priv)

Update Ethernet data input path (i.e. specify where to pass the input buffer)

Note: After install driver, Ethernet still don't know where to deliver the input buffer. In fact, this API registers a callback function which get invoked when Ethernet received new packets.

Parameters

- **`hdl`** –**[in]** handle of Ethernet driver
- **`stack_input`** –**[in]** function pointer, which does the actual process on incoming packets
- **`priv`** –**[in]** private resource, which gets passed to `stack_input` callback without any modification

Returns

- ESP_OK: update input path successfully
- ESP_ERR_INVALID_ARG: update input path failed because of some invalid argument
- ESP_FAIL: update input path failed because some other error occurred

esp_err_t **`esp_eth_transmit`** (*esp_eth_handle_t* hdl, void *buf, size_t length)

General Transmit.

Parameters

- **`hdl`** –**[in]** handle of Ethernet driver
- **`buf`** –**[in]** buffer of the packet to transfer
- **`length`** –**[in]** length of the buffer to transfer

Returns

- ESP_OK: transmit frame buffer successfully
- ESP_ERR_INVALID_ARG: transmit frame buffer failed because of some invalid argument
- ESP_ERR_INVALID_STATE: invalid driver state (e.i. driver is not started)
- ESP_ERR_TIMEOUT: transmit frame buffer failed because HW was not get available in predefined period
- ESP_FAIL: transmit frame buffer failed because some other error occurred

esp_err_t **`esp_eth_transmit_vargs`** (*esp_eth_handle_t* hdl, uint32_t argc, ...)

Special Transmit with variable number of arguments.

Parameters

- **`hdl`** –**[in]** handle of Ethernet driver
- **`argc`** –**[in]** number variable arguments
- **`...`** –variable arguments

Returns

- ESP_OK: transmit successfull
- ESP_ERR_INVALID_STATE: invalid driver state (e.i. driver is not started)

- ESP_ERR_TIMEOUT: transmit frame buffer failed because HW was not get available in predefined period
- ESP_FAIL: transmit frame buffer failed because some other error occurred

esp_err_t **esp_eth_ioctl**(*esp_eth_handle_t* hdl, *esp_eth_io_cmd_t* cmd, void *data)

Misc IO function of Etherent driver.

The following common IO control commands are supported:

- `ETH_CMD_S_MAC_ADDR` sets Ethernet interface MAC address. `data` argument is pointer to MAC address buffer with expected size of 6 bytes.
- `ETH_CMD_G_MAC_ADDR` gets Ethernet interface MAC address. `data` argument is pointer to a buffer to which MAC address is to be copied. The buffer size must be at least 6 bytes.
- `ETH_CMD_S_PHY_ADDR` sets PHY address in range of <0-31>. `data` argument is pointer to memory of uint32_t datatype from where the configuration option is read.
- `ETH_CMD_G_PHY_ADDR` gets PHY address. `data` argument is pointer to memory of uint32_t datatype to which the PHY address is to be stored.
- `ETH_CMD_S_AUTONEGO` enables or disables Ethernet link speed and duplex mode autonegotiation. `data` argument is pointer to memory of bool datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped.
- `ETH_CMD_G_AUTONEGO` gets current configuration of the Ethernet link speed and duplex mode autonegotiation. `data` argument is pointer to memory of bool datatype to which the current configuration is to be stored.
- `ETH_CMD_S_SPEED` sets the Ethernet link speed. `data` argument is pointer to memory of eth_speed_t datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped and auto-negotiation disabled.
- `ETH_CMD_G_SPEED` gets current Ethernet link speed. `data` argument is pointer to memory of eth_speed_t datatype to which the speed is to be stored.
- `ETH_CMD_S_PROMISCUOUS` sets/resets Ethernet interface promiscuous mode. `data` argument is pointer to memory of bool datatype from which the configuration option is read.
- `ETH_CMD_S_FLOW_CTRL` sets/resets Ethernet interface flow control. `data` argument is pointer to memory of bool datatype from which the configuration option is read.
- `ETH_CMD_S_DUPLEX_MODE` sets the Ethernet duplex mode. `data` argument is pointer to memory of eth_duplex_t datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped and auto-negotiation disabled.
- `ETH_CMD_G_DUPLEX_MODE` gets current Ethernet link duplex mode. `data` argument is pointer to memory of eth_duplex_t datatype to which the duplex mode is to be stored.
- `ETH_CMD_S_PHY_LOOPBACK` sets/resets PHY to/from loopback mode. `data` argument is pointer to memory of bool datatype from which the configuration option is read.
- Note that additional control commands may be available for specific MAC or PHY chips. Please consult specific MAC or PHY documentation or driver code.

Parameters

- **hdl** –**[in]** handle of Ethernet driver
- **cmd** –**[in]** IO control command
- **data** –**[inout]** address of data for `set` command or address where to store the data when used with `get` command

Returns

- ESP_OK: process io command successfully
- ESP_ERR_INVALID_ARG: process io command failed because of some invalid argument
- ESP_FAIL: process io command failed because some other error occurred
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported

esp_err_t **esp_eth_increase_reference**(*esp_eth_handle_t* hdl)

Increase Ethernet driver reference.

Note: Ethernet driver handle can be obtained by os timer, netif, etc. It's dangerous when thread A is using Ethernet but thread B uninstall the driver. Using reference counter can prevent such risk, but care should be taken, when you obtain Ethernet driver, this API must be invoked so that the driver won't be uninstalled during your using time.

Parameters **hdl** –**[in]** handle of Ethernet driver
Returns
- ESP_OK: increase reference successfully
- ESP_ERR_INVALID_ARG: increase reference failed because of some invalid argument

esp_err_t **esp_eth_decrease_reference**(*esp_eth_handle_t* hdl)

Decrease Ethernet driver reference.

Parameters **hdl** –**[in]** handle of Ethernet driver
Returns
- ESP_OK: increase reference successfully
- ESP_ERR_INVALID_ARG: increase reference failed because of some invalid argument

Structures

struct **esp_eth_config_t**

Configuration of Ethernet driver.

Public Members

esp_eth_mac_t ***mac**

Ethernet MAC object.

esp_eth_phy_t ***phy**

Ethernet PHY object.

uint32_t **check_link_period_ms**

Period time of checking Ethernet link status.

esp_err_t (***stack_input**)(*esp_eth_handle_t* eth_handle, uint8_t *buffer, uint32_t length, void *priv)

Input frame buffer to user's stack.

Param eth_handle **[in]** handle of Ethernet driver
Param buffer **[in]** frame buffer that will get input to upper stack
Param length **[in]** length of the frame buffer
Return
- ESP_OK: input frame buffer to upper stack successfully
- ESP_FAIL: error occurred when inputting buffer to upper stack

esp_err_t (***on_lowlevel_init_done**)(*esp_eth_handle_t* eth_handle)

Callback function invoked when lowlevel initialization is finished.

Param eth_handle **[in]** handle of Ethernet driver
Return
- ESP_OK: process extra lowlevel initialization successfully
- ESP_FAIL: error occurred when processing extra lowlevel initialization

esp_err_t (*`on_lowlevel_deinit_done`)(*esp_eth_handle_t* eth_handle)

Callback function invoked when lowlevel deinitialization is finished.

Param eth_handle [in] handle of Ethernet driver
Return
- ESP_OK: process extra lowlevel deinitialization successfully
- ESP_FAIL: error occurred when processing extra lowlevel deinitialization

esp_err_t (*`read_phy_reg`)(*esp_eth_handle_t* eth_handle, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)

Read PHY register.

Note: Usually the PHY register read/write function is provided by MAC (SMI interface), but if the PHY device is managed by other interface (e.g. I2C), then user needs to implement the corresponding read/write. Setting this to NULL means your PHY device is managed by MAC' s SMI interface.

Param eth_handle [in] handle of Ethernet driver
Param phy_addr [in] PHY chip address (0~31)
Param phy_reg [in] PHY register index code
Param reg_value [out] PHY register value
Return
- ESP_OK: read PHY register successfully
- ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
- ESP_ERR_TIMEOUT: read PHY register failed because of timeout
- ESP_FAIL: read PHY register failed because some other error occurred

esp_err_t (*`write_phy_reg`)(*esp_eth_handle_t* eth_handle, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)

Write PHY register.

Note: Usually the PHY register read/write function is provided by MAC (SMI interface), but if the PHY device is managed by other interface (e.g. I2C), then user needs to implement the corresponding read/write. Setting this to NULL means your PHY device is managed by MAC' s SMI interface.

Param eth_handle [in] handle of Ethernet driver
Param phy_addr [in] PHY chip address (0~31)
Param phy_reg [in] PHY register index code
Param reg_value [in] PHY register value
Return
- ESP_OK: write PHY register successfully
- ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
- ESP_ERR_TIMEOUT: write PHY register failed because of timeout
- ESP_FAIL: write PHY register failed because some other error occurred

struct **`esp_eth_phy_reg_rw_data_t`**

Data structure to Read/Write PHY register via ioctl API.

Public Members

uint32_t **`reg_addr`**

PHY register address

uint32_t ***reg_value_p**

Pointer to a memory where the register value is read/written

Macros

ETH_DEFAULT_CONFIG(emac, ephy)

Default configuration for Ethernet driver.

Type Definitions

typedef void ***esp_eth_handle_t**

Handle of Ethernet driver.

Enumerations

enum **esp_eth_io_cmd_t**

Command list for ioctl API.

Values:

enumerator **ETH_CMD_G_MAC_ADDR**

Get MAC address

enumerator **ETH_CMD_S_MAC_ADDR**

Set MAC address

enumerator **ETH_CMD_G_PHY_ADDR**

Get PHY address

enumerator **ETH_CMD_S_PHY_ADDR**

Set PHY address

enumerator **ETH_CMD_G_AUTONEGO**

Get PHY Auto Negotiation

enumerator **ETH_CMD_S_AUTONEGO**

Set PHY Auto Negotiation

enumerator **ETH_CMD_G_SPEED**

Get Speed

enumerator **ETH_CMD_S_SPEED**

Set Speed

enumerator **ETH_CMD_S_PROMISCUOUS**

Set promiscuous mode

enumerator **ETH_CMD_S_FLOW_CTRL**

Set flow control

enumerator **ETH_CMD_G_DUPLEX_MODE**

Get Duplex mode

enumerator **ETH_CMD_S_DUPLEX_MODE**

Set Duplex mode

enumerator **ETH_CMD_S_PHY_LOOPBACK**

Set PHY loopback

enumerator **ETH_CMD_READ_PHY_REG**

Read PHY register

enumerator **ETH_CMD_WRITE_PHY_REG**

Write PHY register

enumerator **ETH_CMD_CUSTOM_MAC_CMDS**

enumerator **ETH_CMD_CUSTOM_PHY_CMDS**

Header File

- components/esp_eth/include/esp_eth_com.h

Structures

struct **esp_eth_mediator_s**

Ethernet mediator.

Public Members

esp_err_t (***phy_reg_read**)(*esp_eth_mediator_t* *eth, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)

Read PHY register.

Param eth **[in]** mediator of Ethernet driver
Param phy_addr **[in]** PHY Chip address (0~31)
Param phy_reg **[in]** PHY register index code
Param reg_value **[out]** PHY register value
Return

- ESP_OK: read PHY register successfully
- ESP_FAIL: read PHY register failed because some error occurred

esp_err_t (***phy_reg_write**)(*esp_eth_mediator_t* *eth, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)

Write PHY register.

Param eth **[in]** mediator of Ethernet driver
Param phy_addr **[in]** PHY Chip address (0~31)
Param phy_reg **[in]** PHY register index code
Param reg_value **[in]** PHY register value
Return

- ESP_OK: write PHY register successfully

- ESP_FAIL: write PHY register failed because some error occurred

esp_err_t (*`stack_input`)(*esp_eth_mediator_t* *eth, uint8_t *buffer, uint32_t length)

Deliver packet to upper stack.

Param eth **[in]** mediator of Ethernet driver
Param buffer **[in]** packet buffer
Param length **[in]** length of the packet
Return
- ESP_OK: deliver packet to upper stack successfully
- ESP_FAIL: deliver packet failed because some error occurred

esp_err_t (*`on_state_changed`)(*esp_eth_mediator_t* *eth, *esp_eth_state_t* state, void *args)

Callback on Ethernet state changed.

Param eth **[in]** mediator of Ethernet driver
Param state **[in]** new state
Param args **[in]** optional argument for the new state
Return
- ESP_OK: process the new state successfully
- ESP_FAIL: process the new state failed because some error occurred

Type Definitions

typedef struct *esp_eth_mediator_s* **`esp_eth_mediator_t`**

Ethernet mediator.

Enumerations

enum **`esp_eth_state_t`**

Ethernet driver state.

Values:

enumerator **`ETH_STATE_LLINIT`**

Lowlevel init done

enumerator **`ETH_STATE_DEINIT`**

Deinit done

enumerator **`ETH_STATE_LINK`**

Link status changed

enumerator **`ETH_STATE_SPEED`**

Speed updated

enumerator **`ETH_STATE_DUPLEX`**

Duplex updated

enumerator **`ETH_STATE_PAUSE`**

Pause ability updated

enum **eth_event_t**

Ethernet event declarations.

Values:

enumerator **ETHERNET_EVENT_START**

Ethernet driver start

enumerator **ETHERNET_EVENT_STOP**

Ethernet driver stop

enumerator **ETHERNET_EVENT_CONNECTED**

Ethernet got a valid link

enumerator **ETHERNET_EVENT_DISCONNECTED**

Ethernet lost a valid link

Header File

- components/esp_eth/include/esp_eth_mac.h

Functions

esp_eth_mac_t ***esp_eth_mac_new_esp32** (const *eth_esp32_emac_config_t* *esp32_config, const *eth_mac_config_t* *config)

Create ESP32 Ethernet MAC instance.

Parameters

- **esp32_config** –EMAC specific configuration
- **config** –Ethernet MAC configuration

Returns

- instance: create MAC instance successfully
- NULL: create MAC instance failed because some error occurred

Unions

union **eth_mac_clock_config_t**

#include <esp_eth_mac.h> Ethernet MAC Clock Configuration.

Public Members

struct *eth_mac_clock_config_t*::[anonymous] **mii**

EMAC MII Clock Configuration

emac_rmii_clock_mode_t **clock_mode**

RMII Clock Mode Configuration

emac_rmii_clock_gpio_t **clock_gpio**

RMII Clock GPIO Configuration

struct *eth_mac_clock_config_t*::[anonymous] **rmii**

EMAC RMII Clock Configuration

Structures

struct **esp_eth_mac_s**

Ethernet MAC.

Public Members

esp_err_t (***set_mediator**)(*esp_eth_mac_t* *mac, *esp_eth_mediator_t* *eth)

Set mediator for Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Param eth **[in]** Ethernet mediator
Return
- ESP_OK: set mediator for Ethernet MAC successfully
- ESP_ERR_INVALID_ARG: set mediator for Ethernet MAC failed because of invalid argument

esp_err_t (***init**)(*esp_eth_mac_t* *mac)

Initialize Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Return
- ESP_OK: initialize Ethernet MAC successfully
- ESP_ERR_TIMEOUT: initialize Ethernet MAC failed because of timeout
- ESP_FAIL: initialize Ethernet MAC failed because some other error occurred

esp_err_t (***deinit**)(*esp_eth_mac_t* *mac)

Deinitialize Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Return
- ESP_OK: deinitialize Ethernet MAC successfully
- ESP_FAIL: deinitialize Ethernet MAC failed because some error occurred

esp_err_t (***start**)(*esp_eth_mac_t* *mac)

Start Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Return
- ESP_OK: start Ethernet MAC successfully
- ESP_FAIL: start Ethernet MAC failed because some other error occurred

esp_err_t (***stop**)(*esp_eth_mac_t* *mac)

Stop Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Return
- ESP_OK: stop Ethernet MAC successfully
- ESP_FAIL: stop Ethernet MAC failed because some error occurred

esp_err_t (***transmit**)(*esp_eth_mac_t* *mac, uint8_t *buf, uint32_t length)

Transmit packet from Ethernet MAC.

Note: Returned error codes may differ for each specific MAC chip.

Param mac **[in]** Ethernet MAC instance
Param buf **[in]** packet buffer to transmit
Param length **[in]** length of packet
Return
- ESP_OK: transmit packet successfully
- ESP_ERR_INVALID_SIZE: number of actually sent bytes differs to expected
- ESP_FAIL: transmit packet failed because some other error occurred

esp_err_t (*`transmit_vargs`)(*esp_eth_mac_t* *mac, uint32_t argc, va_list args)

Transmit packet from Ethernet MAC constructed with special parameters at Layer2.

Note: Typical intended use case is to make possible to construct a frame from multiple higher layer buffers without a need of buffer reallocations. However, other use cases are not limited.

Note: Returned error codes may differ for each specific MAC chip.

Param mac **[in]** Ethernet MAC instance
Param argc **[in]** number variable arguments
Param args **[in]** variable arguments
Return
- ESP_OK: transmit packet successfully
- ESP_ERR_INVALID_SIZE: number of actually sent bytes differs to expected
- ESP_FAIL: transmit packet failed because some other error occurred

esp_err_t (*`receive`)(*esp_eth_mac_t* *mac, uint8_t *buf, uint32_t *length)

Receive packet from Ethernet MAC.

Note: Memory of buf is allocated in the Layer2, make sure it get free after process.

Note: Before this function got invoked, the value of "length" should set by user, equals the size of buffer. After the function returned, the value of "length" means the real length of received data.

Param mac **[in]** Ethernet MAC instance
Param buf **[out]** packet buffer which will preserve the received frame
Param length **[out]** length of the received packet
Return
- ESP_OK: receive packet successfully
- ESP_ERR_INVALID_ARG: receive packet failed because of invalid argument
- ESP_ERR_INVALID_SIZE: input buffer size is not enough to hold the incoming data. in this case, value of returned "length" indicates the real size of incoming data.
- ESP_FAIL: receive packet failed because some other error occurred

esp_err_t (*`read_phy_reg`)(*esp_eth_mac_t* *mac, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)

Read PHY register.

Param mac **[in]** Ethernet MAC instance
Param phy_addr **[in]** PHY chip address (0~31)
Param phy_reg **[in]** PHY register index code
Param reg_value **[out]** PHY register value

Return
- ESP_OK: read PHY register successfully
- ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
- ESP_ERR_INVALID_STATE: read PHY register failed because of wrong state of MAC
- ESP_ERR_TIMEOUT: read PHY register failed because of timeout
- ESP_FAIL: read PHY register failed because some other error occurred

esp_err_t (*`write_phy_reg`)(*esp_eth_mac_t* *mac, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)

Write PHY register.

Param mac **[in]** Ethernet MAC instance
Param phy_addr **[in]** PHY chip address (0~31)
Param phy_reg **[in]** PHY register index code
Param reg_value **[in]** PHY register value
Return
- ESP_OK: write PHY register successfully
- ESP_ERR_INVALID_STATE: write PHY register failed because of wrong state of MAC
- ESP_ERR_TIMEOUT: write PHY register failed because of timeout
- ESP_FAIL: write PHY register failed because some other error occurred

esp_err_t (*`set_addr`)(*esp_eth_mac_t* *mac, uint8_t *addr)

Set MAC address.

Param mac **[in]** Ethernet MAC instance
Param addr **[in]** MAC address
Return
- ESP_OK: set MAC address successfully
- ESP_ERR_INVALID_ARG: set MAC address failed because of invalid argument
- ESP_FAIL: set MAC address failed because some other error occurred

esp_err_t (*`get_addr`)(*esp_eth_mac_t* *mac, uint8_t *addr)

Get MAC address.

Param mac **[in]** Ethernet MAC instance
Param addr **[out]** MAC address
Return
- ESP_OK: get MAC address successfully
- ESP_ERR_INVALID_ARG: get MAC address failed because of invalid argument
- ESP_FAIL: get MAC address failed because some other error occurred

esp_err_t (*`set_speed`)(*esp_eth_mac_t* *mac, eth_speed_t speed)

Set speed of MAC.

Param ma:c **[in]** Ethernet MAC instance
Param speed **[in]** MAC speed
Return
- ESP_OK: set MAC speed successfully
- ESP_ERR_INVALID_ARG: set MAC speed failed because of invalid argument
- ESP_FAIL: set MAC speed failed because some other error occurred

esp_err_t (*`set_duplex`)(*esp_eth_mac_t* *mac, eth_duplex_t duplex)

Set duplex mode of MAC.

Param mac **[in]** Ethernet MAC instance
Param duplex **[in]** MAC duplex

Return
- ESP_OK: set MAC duplex mode successfully
- ESP_ERR_INVALID_ARG: set MAC duplex failed because of invalid argument
- ESP_FAIL: set MAC duplex failed because some other error occurred

esp_err_t (*`set_link`)(*esp_eth_mac_t* *mac, eth_link_t link)

Set link status of MAC.

Param mac **[in]** Ethernet MAC instance
Param link **[in]** Link status
Return
- ESP_OK: set link status successfully
- ESP_ERR_INVALID_ARG: set link status failed because of invalid argument
- ESP_FAIL: set link status failed because some other error occurred

esp_err_t (*`set_promiscuous`)(*esp_eth_mac_t* *mac, bool enable)

Set promiscuous of MAC.

Param mac **[in]** Ethernet MAC instance
Param enable **[in]** set true to enable promiscuous mode; set false to disable promiscuous mode
Return
- ESP_OK: set promiscuous mode successfully
- ESP_FAIL: set promiscuous mode failed because some error occurred

esp_err_t (*`enable_flow_ctrl`)(*esp_eth_mac_t* *mac, bool enable)

Enable flow control on MAC layer or not.

Param mac **[in]** Ethernet MAC instance
Param enable **[in]** set true to enable flow control; set false to disable flow control
Return
- ESP_OK: set flow control successfully
- ESP_FAIL: set flow control failed because some error occurred

esp_err_t (*`set_peer_pause_ability`)(*esp_eth_mac_t* *mac, uint32_t ability)

Set the PAUSE ability of peer node.

Param mac **[in]** Ethernet MAC instance
Param ability **[in]** zero indicates that pause function is supported by link partner; non-zero indicates that pause function is not supported by link partner
Return
- ESP_OK: set peer pause ability successfully
- ESP_FAIL: set peer pause ability failed because some error occurred

esp_err_t (*`custom_ioctl`)(*esp_eth_mac_t* *mac, uint32_t cmd, void *data)

Custom IO function of MAC driver. This function is intended to extend common options of esp_eth_ioctl to cover specifics of MAC chip.

Note: This function may not be assigned when the MAC chip supports only most common set of configuration options.

Param mac **[in]** Ethernet MAC instance
Param cmd **[in]** IO control command
Param data **[inout]** address of data for `set` command or address where to store the data when used with `get` command
Return
- ESP_OK: process io command successfully

- ESP_ERR_INVALID_ARG: process io command failed because of some invalid argument
- ESP_FAIL: process io command failed because some other error occurred
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported

esp_err_t (*`del`)(*esp_eth_mac_t* *mac)

Free memory of Ethernet MAC.

Param mac **[in]** Ethernet MAC instance
Return
- ESP_OK: free Ethernet MAC instance successfully
- ESP_FAIL: free Ethernet MAC instance failed because some error occurred

struct **`eth_mac_config_t`**

Configuration of Ethernet MAC object.

Public Members

uint32_t **`sw_reset_timeout_ms`**

Software reset timeout value (Unit: ms)

uint32_t **`rx_task_stack_size`**

Stack size of the receive task

uint32_t **`rx_task_prio`**

Priority of the receive task

uint32_t **`flags`**

Flags that specify extra capability for mac driver

struct **`eth_esp32_emac_config_t`**

EMAC specific configuration.

Public Members

int **`smi_mdc_gpio_num`**

SMI MDC GPIO number, set to -1 could bypass the SMI GPIO configuration

int **`smi_mdio_gpio_num`**

SMI MDIO GPIO number, set to -1 could bypass the SMI GPIO configuration

eth_data_interface_t **`interface`**

EMAC Data interface to PHY (MII/RMII)

eth_mac_clock_config_t **`clock_config`**

EMAC Interface clock configuration

eth_mac_dma_burst_len_t **`dma_burst_len`**

EMAC DMA burst length for both Tx and Rx

Macros

ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE

MAC driver can work when cache is disabled

ETH_MAC_FLAG_PIN_TO_CORE

Pin MAC task to the CPU core where driver installation happened

ETH_MAC_DEFAULT_CONFIG()

Default configuration for Ethernet MAC object.

ETH_ESP32_EMAC_DEFAULT_CONFIG()

Default ESP32's EMAC specific configuration.

Type Definitions

typedef struct *esp_eth_mac_s* **esp_eth_mac_t**

Ethernet MAC.

Enumerations

enum **emac_rmii_clock_mode_t**

RMII Clock Mode Options.

Values:

enumerator **EMAC_CLK_DEFAULT**

Default values configured using Kconfig are going to be used when "Default" selected.

enumerator **EMAC_CLK_EXT_IN**

Input RMII Clock from external. EMAC Clock GPIO number needs to be configured when this option is selected.

Note: MAC will get RMII clock from outside. Note that ESP32 only supports GPIO0 to input the RMII clock.

---

enumerator **EMAC_CLK_OUT**

Output RMII Clock from internal APLL Clock. EMAC Clock GPIO number needs to be configured when this option is selected.

enum **emac_rmii_clock_gpio_t**

RMII Clock GPIO number Options.

Values:

enumerator **EMAC_CLK_IN_GPIO**

MAC will get RMII clock from outside at this GPIO.

Note: ESP32 only supports GPIO0 to input the RMII clock.

---

enumerator **EMAC_APPL_CLK_OUT_GPIO**

Output RMII Clock from internal APLL Clock available at GPIO0.

Note: GPIO0 can be set to output a pre-divided PLL clock (test only!). Enabling this option will configure GPIO0 to output a 50MHz clock. In fact this clock doesn't have directly relationship with EMAC peripheral. Sometimes this clock won't work well with your PHY chip. You might need to add some extra devices after GPIO0 (e.g. inverter). Note that outputting RMII clock on GPIO0 is an experimental practice. If you want the Ethernet to work with WiFi, don't select GPIO0 output mode for stability.

---

enumerator **EMAC_CLK_OUT_GPIO**

Output RMII Clock from internal APLL Clock available at GPIO16.

enumerator **EMAC_CLK_OUT_180_GPIO**

Inverted Output RMII Clock from internal APLL Clock available at GPIO17.

Header File

- components/esp_eth/include/esp_eth_phy.h

Functions

esp_eth_phy_t ***esp_eth_phy_new_ip101** (const *eth_phy_config_t* *config)

Create a PHY instance of IP101.

Parameters **config** –**[in]** configuration of PHY

Returns

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

esp_eth_phy_t ***esp_eth_phy_new_rtl8201** (const *eth_phy_config_t* *config)

Create a PHY instance of RTL8201.

Parameters **config** –**[in]** configuration of PHY

Returns

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

esp_eth_phy_t ***esp_eth_phy_new_lan87xx** (const *eth_phy_config_t* *config)

Create a PHY instance of LAN87xx.

Parameters **config** –**[in]** configuration of PHY

Returns

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

esp_eth_phy_t ***esp_eth_phy_new_dp83848** (const *eth_phy_config_t* *config)

Create a PHY instance of DP83848.

Parameters **config** –**[in]** configuration of PHY

Returns

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

esp_eth_phy_t ***esp_eth_phy_new_ksz80xx** (const *eth_phy_config_t* *config)

Create a PHY instance of KSZ80xx.

The phy model from the KSZ80xx series is detected automatically. If the driver is unable to detect a supported model, `NULL` is returned.

Currently, the following models are supported: KSZ8001, KSZ8021, KSZ8031, KSZ8041, KSZ8051, KSZ8061, KSZ8081, KSZ8091

Parameters **config** –**[in]** configuration of PHY
Returns
- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

Structures

struct **esp_eth_phy_s**

Ethernet PHY.

Public Members

esp_err_t (***set_mediator**)(*esp_eth_phy_t* *phy, *esp_eth_mediator_t* *mediator)

Set mediator for PHY.

Param phy **[in]** Ethernet PHY instance
Param mediator **[in]** mediator of Ethernet driver
Return
- ESP_OK: set mediator for Ethernet PHY instance successfully
- ESP_ERR_INVALID_ARG: set mediator for Ethernet PHY instance failed because of some invalid arguments

esp_err_t (***reset**)(*esp_eth_phy_t* *phy)

Software Reset Ethernet PHY.

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: reset Ethernet PHY successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

esp_err_t (***reset_hw**)(*esp_eth_phy_t* *phy)

Hardware Reset Ethernet PHY.

Note: Hardware reset is mostly done by pull down and up PHY’ s nRST pin

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: reset Ethernet PHY successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

esp_err_t (***init**)(*esp_eth_phy_t* *phy)

Initialize Ethernet PHY.

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: initialize Ethernet PHY successfully
- ESP_FAIL: initialize Ethernet PHY failed because some error occurred

esp_err_t (***deinit**)(*esp_eth_phy_t* *phy)

Deinitialize Ethernet PHY.

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: deinitialize Ethernet PHY successfully
- ESP_FAIL: deinitialize Ethernet PHY failed because some error occurred

esp_err_t (*`autonego_ctrl`)(*esp_eth_phy_t* *phy, *eth_phy_autoneg_cmd_t* cmd, bool *autonego_en_stat)

Configure auto negotiation.

Param phy **[in]** Ethernet PHY instance
Param cmd **[in]** Configuration command, it is possible to Enable (restart), Disable or get current status of PHY auto negotiation
Param autonego_en_stat **[out]** Address where to store current status of auto negotiation configuration
Return
- ESP_OK: restart auto negotiation successfully
- ESP_FAIL: restart auto negotiation failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid command

esp_err_t (*`get_link`)(*esp_eth_phy_t* *phy)

Get Ethernet PHY link status.

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: get Ethernet PHY link status successfully
- ESP_FAIL: get Ethernet PHY link status failed because some error occurred

esp_err_t (*`set_link`)(*esp_eth_phy_t* *phy, eth_link_t link)

Set Ethernet PHY link status.

Param phy **[in]** Ethernet PHY instance
Param link **[in]** new link status
Return
- ESP_OK: set Ethernet PHY link status successfully
- ESP_FAIL: set Ethernet PHY link status failed because some error occurred

esp_err_t (*`pwrctl`)(*esp_eth_phy_t* *phy, bool enable)

Power control of Ethernet PHY.

Param phy **[in]** Ethernet PHY instance
Param enable **[in]** set true to power on Ethernet PHY; ser false to power off Ethernet PHY
Return
- ESP_OK: control Ethernet PHY power successfully
- ESP_FAIL: control Ethernet PHY power failed because some error occurred

esp_err_t (*`set_addr`)(*esp_eth_phy_t* *phy, uint32_t addr)

Set PHY chip address.

Param phy **[in]** Ethernet PHY instance
Param addr **[in]** PHY chip address
Return
- ESP_OK: set Ethernet PHY address successfully
- ESP_FAIL: set Ethernet PHY address failed because some error occurred

esp_err_t (*`get_addr`)(*esp_eth_phy_t* *phy, uint32_t *addr)

Get PHY chip address.

Param phy **[in]** Ethernet PHY instance
Param addr **[out]** PHY chip address

Return
- ESP_OK: get Ethernet PHY address successfully
- ESP_ERR_INVALID_ARG: get Ethernet PHY address failed because of invalid argument

esp_err_t (*`advertise_pause_ability`)(*esp_eth_phy_t* *phy, uint32_t ability)

Advertise pause function supported by MAC layer.

Param phy **[in]** Ethernet PHY instance
Param addr **[out]** Pause ability
Return
- ESP_OK: Advertise pause ability successfully
- ESP_ERR_INVALID_ARG: Advertise pause ability failed because of invalid argument

esp_err_t (*`loopback`)(*esp_eth_phy_t* *phy, bool enable)

Sets the PHY to loopback mode.

Param phy **[in]** Ethernet PHY instance
Param enable **[in]** enables or disables PHY loopback
Return
- ESP_OK: PHY instance loopback mode has been configured successfully
- ESP_FAIL: PHY instance loopback configuration failed because some error occurred

esp_err_t (*`set_speed`)(*esp_eth_phy_t* *phy, eth_speed_t speed)

Sets PHY speed mode.

Note: Autonegotiation feature needs to be disabled prior to calling this function for the new setting to be applied

---

Param phy **[in]** Ethernet PHY instance
Param speed **[in]** Speed mode to be set
Return
- ESP_OK: PHY instance speed mode has been configured successfully
- ESP_FAIL: PHY instance speed mode configuration failed because some error occurred

esp_err_t (*`set_duplex`)(*esp_eth_phy_t* *phy, eth_duplex_t duplex)

Sets PHY duplex mode.

Note: Autonegotiation feature needs to be disabled prior to calling this function for the new setting to be applied

---

Param phy **[in]** Ethernet PHY instance
Param duplex **[in]** Duplex mode to be set
Return
- ESP_OK: PHY instance duplex mode has been configured successfully
- ESP_FAIL: PHY instance duplex mode configuration failed because some error occurred

esp_err_t (*`custom_ioctl`)(*esp_eth_phy_t* *phy, uint32_t cmd, void *data)

Custom IO function of PHY driver. This function is intended to extend common options of esp_eth_ioctl to cover specifics of PHY chip.

Note: This function may not be assigned when the PHY chip supports only most common set of configuration options.

Param phy **[in]** Ethernet PHY instance
Param cmd **[in]** IO control command
Param data **[inout]** address of data for `set` command or address where to store the data when used with `get` command
Return
- ESP_OK: process io command successfully
- ESP_ERR_INVALID_ARG: process io command failed because of some invalid argument
- ESP_FAIL: process io command failed because some other error occurred
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported

esp_err_t (***del**)(*esp_eth_phy_t* *phy)

Free memory of Ethernet PHY instance.

Param phy **[in]** Ethernet PHY instance
Return
- ESP_OK: free PHY instance successfully
- ESP_FAIL: free PHY instance failed because some error occurred

struct **eth_phy_config_t**

Ethernet PHY configuration.

Public Members

int32_t **phy_addr**

PHY address, set -1 to enable PHY address detection at initialization stage

uint32_t **reset_timeout_ms**

Reset timeout value (Unit: ms)

uint32_t **autonego_timeout_ms**

Auto-negotiation timeout value (Unit: ms)

int **reset_gpio_num**

Reset GPIO number, -1 means no hardware reset

Macros

ESP_ETH_PHY_ADDR_AUTO

ETH_PHY_DEFAULT_CONFIG()

Default configuration for Ethernet PHY object.

Type Definitions

typedef struct *esp_eth_phy_s* **esp_eth_phy_t**

Ethernet PHY.

Enumerations

enum **eth_phy_autoneg_cmd_t**

Auto-negotiation controll commands.

Values:

enumerator **ESP_ETH_PHY_AUTONEGO_RESTART**

enumerator **ESP_ETH_PHY_AUTONEGO_EN**

enumerator **ESP_ETH_PHY_AUTONEGO_DIS**

enumerator **ESP_ETH_PHY_AUTONEGO_G_STAT**

Header File

- components/esp_eth/include/esp_eth_phy_802_3.h

Functions

esp_err_t **esp_eth_phy_802_3_set_mediator**(*phy_802_3_t* *phy_802_3, *esp_eth_mediator_t* *eth)

Set Ethernet mediator.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **eth** –Ethernet mediator pointer

Returns
- ESP_OK: Ethermet mediator set successfuly
- ESP_ERR_INVALID_ARG: if `eth` is `NULL`

esp_err_t **esp_eth_phy_802_3_reset**(*phy_802_3_t* *phy_802_3)

Reset PHY.

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY reset successfuly
- ESP_FAIL: reset Ethernet PHY failed because some error occured

esp_err_t **esp_eth_phy_802_3_autonego_ctrl**(*phy_802_3_t* *phy_802_3, *eth_phy_autoneg_cmd_t* cmd, bool *autonego_en_stat)

Control autonegotiation mode of Ethernet PHY.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **cmd** –autonegotiation command enumeration
- **autonego_en_stat** –**[out]** autonegotiation enabled flag

Returns
- ESP_OK: Ethernet PHY autonegotiation configured successfuly
- ESP_FAIL: Ethernet PHY autonegotiation configuration fail because some error occured
- ESP_ERR_INVALID_ARG: invalid value of `cmd`

esp_err_t **esp_eth_phy_802_3_pwrctl**(*phy_802_3_t* *phy_802_3, bool enable)

Power control of Ethernet PHY.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **enable** –set true to power ON Ethernet PHY; set false to power OFF Ethernet PHY

Returns
- ESP_OK: Ethernet PHY power down mode set successfuly

- ESP_FAIL: Ethernet PHY power up or power down failed because some error occured

esp_err_t **esp_eth_phy_802_3_set_addr**(*phy_802_3_t* *phy_802_3, uint32_t addr)

Set Ethernet PHY address.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **addr** –new PHY address

Returns
- ESP_OK: Ethernet PHY address set

esp_err_t **esp_eth_phy_802_3_get_addr**(*phy_802_3_t* *phy_802_3, uint32_t *addr)

Get Ethernet PHY address.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **addr** –**[out]** Ethernet PHY address

Returns
- ESP_OK: Ethernet PHY address read successfuly
- ESP_ERR_INVALID_ARG: `addr` pointer is `NULL`

esp_err_t **esp_eth_phy_802_3_advertise_pause_ability**(*phy_802_3_t* *phy_802_3, uint32_t ability)

Advertise pause function ability.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **ability** –enable or disable pause ability

Returns
- ESP_OK: pause ability set successfuly
- ESP_FAIL: Advertise pause function ability failed because some error occured

esp_err_t **esp_eth_phy_802_3_loopback**(*phy_802_3_t* *phy_802_3, bool enable)

Set Ethernet PHY loopback mode.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **enable** –set true to enable loopback; set false to disable loopback

Returns
- ESP_OK: Ethernet PHY loopback mode set successfuly
- ESP_FAIL: Ethernet PHY loopback configuration failed because some error occured

esp_err_t **esp_eth_phy_802_3_set_speed**(*phy_802_3_t* *phy_802_3, eth_speed_t speed)

Set Ethernet PHY speed.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **speed** –new speed of Ethernet PHY link

Returns
- ESP_OK: Ethernet PHY speed set successfuly
- ESP_FAIL: Set Ethernet PHY speed failed because some error occured

esp_err_t **esp_eth_phy_802_3_set_duplex**(*phy_802_3_t* *phy_802_3, eth_duplex_t duplex)

Set Ethernet PHY duplex mode.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **duplex** –new duplex mode for Ethernet PHY link

Returns
- ESP_OK: Ethernet PHY duplex mode set successfuly
- ESP_ERR_INVALID_STATE: unable to set duplex mode to Half if loopback is enabled
- ESP_FAIL: Set Ethernet PHY duplex mode failed because some error occured

esp_err_t **esp_eth_phy_802_3_set_link**(*phy_802_3_t* *phy_802_3, eth_link_t link)

Set Ethernet PHY link status.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **link** –new link status

Returns
- ESP_OK: Ethernet PHY link set successfuly

esp_err_t **esp_eth_phy_802_3_init**(*phy_802_3_t* *phy_802_3)

Initialize Ethernet PHY.

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY initialized successfuly

esp_err_t **esp_eth_phy_802_3_deinit**(*phy_802_3_t* *phy_802_3)

Power off Eternet PHY.

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY powered off successfuly

esp_err_t **esp_eth_phy_802_3_del**(*phy_802_3_t* *phy_802_3)

Delete Ethernet PHY infostructure.

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethrnet PHY infostructure deleted

esp_err_t **esp_eth_phy_802_3_reset_hw**(*phy_802_3_t* *phy_802_3, uint32_t reset_assert_us)

Performs hardware reset with specific reset pin assertion time.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **reset_assert_us** –Hardware reset pin assertion time

Returns
- ESP_OK: reset Ethernet PHY successfully

esp_err_t **esp_eth_phy_802_3_detect_phy_addr**(*esp_eth_mediator_t* *eth, int *detected_addr)

Detect PHY address.

Parameters
- **eth** –Mediator of Ethernet driver
- **detected_addr** –**[out]** a valid address after detection

Returns
- ESP_OK: detect phy address successfully
- ESP_ERR_INVALID_ARG: invalid parameter
- ESP_ERR_NOT_FOUND: can't detect any PHY device
- ESP_FAIL: detect phy address failed because some error occurred

esp_err_t **esp_eth_phy_802_3_basic_phy_init**(*phy_802_3_t* *phy_802_3)

Performs basic PHY chip initialization.

Note: It should be called as the first function in PHY specific driver instance

---

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: initialized Ethernet PHY successfully
- ESP_FAIL: initialization of Ethernet PHY failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid argument

- ESP_ERR_NOT_FOUND: PHY device not detected
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

esp_err_t **esp_eth_phy_802_3_basic_phy_deinit** (*phy_802_3_t* *phy_802_3)

Performs basic PHY chip de-initialization.

Note: It should be called as the last function in PHY specific driver instance

Parameters **phy_802_3** –IEEE 802.3 PHY object infostructure

Returns

- ESP_OK: de-initialized Ethernet PHY successfully
- ESP_FAIL: de-initialization of Ethernet PHY failed because some error occurred
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

esp_err_t **esp_eth_phy_802_3_read_oui** (*phy_802_3_t* *phy_802_3, uint32_t *oui)

Reads raw content of OUI field.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **oui** –**[out]** OUI value

Returns

- ESP_OK: OUI field read successfully
- ESP_FAIL: OUI field read failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid `oui` argument
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

esp_err_t **esp_eth_phy_802_3_read_manufac_info** (*phy_802_3_t* *phy_802_3, uint8_t *model, uint8_t *rev)

Reads manufacturer's model and revision number.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **model** –**[out]** Manufacturer's model number (can be NULL when not required)
- **rev** –**[out]** Manufacturer's revision number (can be NULL when not required)

Returns

- ESP_OK: Manufacturer's info read successfully
- ESP_FAIL: Manufacturer's info read failed because some error occurred
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

esp_err_t **esp_eth_phy_802_3_get_mmd_addr** (*phy_802_3_t* *phy_802_3, uint8_t devaddr, uint16_t *mmd_addr)

Reads MDIO device's internal address register.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **mmd_addr** –**[out]** Current address stored in device's register

Returns

- ESP_OK: Address register read successfuly
- ESP_FAIL: Address register read failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits)

esp_err_t **esp_eth_phy_802_3_set_mmd_addr**(*phy_802_3_t* *phy_802_3, uint8_t devaddr, uint16_t mmd_addr)

Write to DIO device's internal address register.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **mmd_addr** –**[out]** New value of MDIO device's address register value

Returns

- ESP_OK: Address register written to successfuly
- ESP_FAIL: Address register write failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits)

esp_err_t **esp_eth_phy_802_3_read_mmd_data**(*phy_802_3_t* *phy_802_3, uint8_t devaddr, *esp_eth_phy_802_3_mmd_func_t* function, uint32_t *data)

Read data of MDIO device's memory at address register.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **function** –MMD function
- **data** –**[out]** Data read from the device's memory

Returns

- ESP_OK: Memory read successfuly
- ESP_FAIL: Memory read failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits) or MMD access function is invalid

esp_err_t **esp_eth_phy_802_3_write_mmd_data**(*phy_802_3_t* *phy_802_3, uint8_t devaddr, *esp_eth_phy_802_3_mmd_func_t* function, uint32_t data)

Write data to MDIO device's memory at address register.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **function** –MMD function
- **data** –**[out]** Data to write to the device's memory

Returns

- ESP_OK: Memory written successfuly
- ESP_FAIL: Memory write failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits) or MMD access function is invalid

esp_err_t **esp_eth_phy_802_3_read_mmd_register**(*phy_802_3_t* *phy_802_3, uint8_t devaddr, uint16_t mmd_addr, uint32_t *data)

Set MMD address to mmd_addr with function MMD_FUNC_NOINCR and read contents to *data.

Parameters

- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **mmd_addr** –Address of MDIO device register
- **data** –**[out]** Data read from the device's memory

Returns

- ESP_OK: Memory read successfuly
- ESP_FAIL: Memory read failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits)

esp_err_t **esp_eth_phy_802_3_write_mmd_register**(*phy_802_3_t* *phy_802_3, uint8_t devaddr, uint16_t mmd_addr, uint32_t data)

Set MMD address to mmd_addr with function MMD_FUNC_NOINCR and write data.

Parameters
- **phy_802_3** –IEEE 802.3 PHY object infostructure
- **devaddr** –Address of MDIO device
- **mmd_addr** –Address of MDIO device register
- **data** –**[out]** Data to write to the device’s memory

Returns
- ESP_OK: Memory written to successfuly
- ESP_FAIL: Memory write failed because of some error occured
- ESP_ERR_INVALID_ARG: Device address provided is out of range (hardware limits device address to 5 bits)

inline *phy_802_3_t* ***esp_eth_phy_into_phy_802_3**(*esp_eth_phy_t* *phy)

Returns address to parent IEEE 802.3 PHY object infostructure.

Parameters **phy** –Ethernet PHY instance
Returns phy_802_3_t*
- address to parent IEEE 802.3 PHY object infostructure

esp_err_t **esp_eth_phy_802_3_obj_config_init**(*phy_802_3_t* *phy_802_3, const *eth_phy_config_t* *config)

Initializes configuration of parent IEEE 802.3 PHY object infostructure.

Parameters
- **phy_802_3** –Address to IEEE 802.3 PHY object infostructure
- **config** –Configuration of the IEEE 802.3 PHY object

Returns
- ESP_OK: configuration initialized successfully
- ESP_ERR_INVALID_ARG: invalid `config` argument

Structures

struct **phy_802_3_t**

IEEE 802.3 PHY object infostructure.

Public Members

esp_eth_phy_t **parent**

Parent Ethernet PHY instance

esp_eth_mediator_t ***eth**

Mediator of Ethernet driver

int **addr**

PHY address

uint32_t **reset_timeout_ms**

Reset timeout value (Unit: ms)

uint32_t **autonego_timeout_ms**

Auto-negotiation timeout value (Unit: ms)

eth_link_t **link_status**

Current Link status

int **reset_gpio_num**

Reset GPIO number, -1 means no hardware reset

Enumerations

enum **esp_eth_phy_802_3_mmd_func_t**

IEEE 802.3 MMD modes enumeration.

Values:

enumerator **MMD_FUNC_ADDRESS**

enumerator **MMD_FUNC_DATA_NOINCR**

enumerator **MMD_FUNC_DATA_RWINCR**

enumerator **MMD_FUNC_DATA_WINCR**

Header File

- components/esp_eth/include/esp_eth_netif_glue.h

Functions

esp_eth_netif_glue_handle_t **esp_eth_new_netif_glue**(*esp_eth_handle_t* eth_hdl)

Create a netif glue for Ethernet driver.

Note: netif glue is used to attach io driver to TCP/IP netif

Parameters **eth_hdl** – Ethernet driver handle
Returns glue object, which inherits esp_netif_driver_base_t

esp_err_t **esp_eth_del_netif_glue**(*esp_eth_netif_glue_handle_t* eth_netif_glue)

Delete netif glue of Ethernet driver.

Parameters **eth_netif_glue** – netif glue
Returns -ESP_OK: delete netif glue successfully

Type Definitions

typedef struct esp_eth_netif_glue_t ***esp_eth_netif_glue_handle_t**

Handle of netif glue - an intermediate layer between netif and Ethernet driver.

Code examples for the Ethernet API are provided in the ethernet directory of ESP-IDF examples.

2.5.3 Thread

Thread

Introduction Thread is a IP-based mesh networking protocol. It's based on the 802.15.4 physical and MAC layer.

Application Examples The openthread directory of ESP-IDF examples contains the following applications:

- The OpenThread interactive shell openthread/ot_cli.
- The Thread border router openthread/ot_br.
- The Thread radio co-processor openthread/ot_rcp.

API Reference For manipulating the Thread network, the OpenThread api shall be used. The OpenThread api docs can be found at the OpenThread official website.

ESP-IDF provides extra apis for launching and managing the OpenThread stack, binding to network interfaces and border routing features.

Header File

- components/openthread/include/esp_openthread.h

Functions

esp_err_t **esp_openthread_init**(const *esp_openthread_platform_config_t* *init_config)

Initializes the full OpenThread stack.

Note: The OpenThread instance will also be initialized in this function.

Parameters **init_config** –**[in]** The initialization configuration.

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_ARG if radio or host connection mode not supported
- ESP_ERR_INVALID_STATE if already initialized

esp_err_t **esp_openthread_launch_mainloop**(void)

Launches the OpenThread main loop.

Note: Thie function will not return unless error happens when running the OpenThread stack.

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_FAIL on other failures

esp_err_t **esp_openthread_deinit**(void)

This function performs OpenThread stack and platform driver deinitialization.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized

otInstance ***esp_openthread_get_instance**(void)

This function acquires the underlying OpenThread instance.

Note: This function can be called on other tasks without lock.

Returns The OpenThread instance pointer

Header File

- components/openthread/include/esp_openthread_types.h

Structures

struct **esp_openthread_mainloop_context_t**

This structure represents a context for a select() based mainloop.

Public Members

fd_set **read_fds**

The read file descriptors

fd_set **write_fds**

The write file descriptors

fd_set **error_fds**

The error file descriptors

int **max_fd**

The max file descriptor

struct timeval **timeout**

The timeout

struct **esp_openthread_uart_config_t**

The uart port config for OpenThread.

Public Members

uart_port_t **port**

UART port number

uart_config_t **uart_config**

UART configuration, see *uart_config_t* docs

int **rx_pin**

UART RX pin

int **tx_pin**

UART TX pin

struct **esp_openthread_radio_config_t**

The OpenThread radio configuration.

Public Members

esp_openthread_radio_mode_t **radio_mode**

The radio mode

esp_openthread_uart_config_t **radio_uart_config**

The uart configuration to RCP

struct **esp_openthread_host_connection_config_t**

The OpenThread host connection configuration.

Public Members

esp_openthread_host_connection_mode_t **host_connection_mode**

The host connection mode

esp_openthread_uart_config_t **host_uart_config**

The uart configuration to host

struct **esp_openthread_port_config_t**

The OpenThread port specific configuration.

Public Members

const char ***storage_partition_name**

The partition for storing OpenThread dataset

uint8_t **netif_queue_size**

The packet queue size for the network interface

uint8_t **task_queue_size**

The task queue size

struct **esp_openthread_platform_config_t**

The OpenThread platform configuration.

Public Members

esp_openthread_radio_config_t **radio_config**

The radio configuration

esp_openthread_host_connection_config_t **host_config**

The host connection configuration

esp_openthread_port_config_t **port_config**

The port configuration

Type Definitions

typedef void (*`esp_openthread_rcp_failure_handler`)(void)

Enumerations

enum `esp_openthread_event_t`

OpenThread event declarations.

Values:

enumerator `OPENTHREAD_EVENT_START`

OpenThread stack start

enumerator `OPENTHREAD_EVENT_STOP`

OpenThread stack stop

enumerator `OPENTHREAD_EVENT_IF_UP`

OpenThread network interface up

enumerator `OPENTHREAD_EVENT_IF_DOWN`

OpenThread network interface down

enumerator `OPENTHREAD_EVENT_GOT_IP6`

OpenThread stack added IPv6 address

enumerator `OPENTHREAD_EVENT_LOST_IP6`

OpenThread stack removed IPv6 address

enumerator `OPENTHREAD_EVENT_MULTICAST_GROUP_JOIN`

OpenThread stack joined IPv6 multicast group

enumerator `OPENTHREAD_EVENT_MULTICAST_GROUP_LEAVE`

OpenThread stack left IPv6 multicast group

enumerator `OPENTHREAD_EVENT_TREL_ADD_IP6`

OpenThread stack added TREL IPv6 address

enumerator `OPENTHREAD_EVENT_TREL_REMOVE_IP6`

OpenThread stack removed TREL IPv6 address

enumerator `OPENTHREAD_EVENT_TREL_MULTICAST_GROUP_JOIN`

OpenThread stack joined TREL IPv6 multicast group

enum `esp_openthread_radio_mode_t`

The radio mode of OpenThread.

Values:

enumerator `RADIO_MODE_NATIVE`

Use the native 15.4 radio

enumerator **RADIO_MODE_UART_RCP**

UART connection to a 15.4 capable radio co-processor (RCP)

enumerator **RADIO_MODE_SPI_RCP**

SPI connection to a 15.4 capable radio co-processor (RCP)

enum **esp_openthread_host_connection_mode_t**

How OpenThread connects to the host.

Values:

enumerator **HOST_CONNECTION_MODE_NONE**

Disable host connection

enumerator **HOST_CONNECTION_MODE_CLI_UART**

CLI UART connection to the host

enumerator **HOST_CONNECTION_MODE_RCP_UART**

RCP UART connection to the host

Header File

- components/openthread/include/esp_openthread_lock.h

Functions

esp_err_t **esp_openthread_lock_init**(void)

This function initializes the OpenThread API lock.

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_STATE if already initialized

void **esp_openthread_lock_deinit**(void)

This function deinitializes the OpenThread API lock.

bool **esp_openthread_lock_acquire**(TickType_t block_ticks)

This functions acquires the OpenThread API lock.

Note: Every OT APIs that takes an otInstance argument MUST be protected with this API lock except that the call site is in OT callbacks.

Parameters **block_ticks** –**[in]** The maxinum number of RTOS ticks to wait for the lock.

Returns

- True on lock acquired
- False on failing to acquire the lock with the timeout.

void **esp_openthread_lock_release**(void)

This function releases the OpenThread API lock.

Header File

- components/openthread/include/esp_openthread_netif_glue.h

Functions

void ***esp_openthread_netif_glue_init**(const *esp_openthread_platform_config_t* *config)

This function initializes the OpenThread network interface glue.

Parameters **config** –**[in]** The platform configuration.

Returns

- glue pointer on success
- NULL on failure

void **esp_openthread_netif_glue_deinit**(void)

This function deinitializes the OpenThread network interface glue.

esp_netif_t ***esp_openthread_get_netif**(void)

This function acquires the OpenThread netif.

Returns The OpenThread netif or NULL if not initialzied.

Macros

ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD()

Default configuration reference of OT esp-netif.

ESP_NETIF_DEFAULT_OPENTHREAD()

Header File

- components/openthread/include/esp_openthread_border_router.h

Functions

void **esp_openthread_set_backbone_netif**(*esp_netif_t* *backbone_netif)

Sets the backbone interface used for border routing.

Note: This function must be called before esp_openthread_init

---

Parameters **backbone_netif** –**[in]** The backbone network interface (WiFi or ethernet)

esp_err_t **esp_openthread_border_router_init**(void)

Initializes the border router features of OpenThread.

Note: Calling this function will make the device behave as an OpenThread border router. Kconfig option CONFIG_OPENTHREAD_BORDER_ROUTER is required.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if feature not supported
- ESP_ERR_INVALID_STATE if already initialized
- ESP_FIAL on other failures

esp_err_t **esp_openthread_border_router_deinit**(void)

Deinitializes the border router features of OpenThread.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized
- ESP_FIAL on other failures

esp_netif_t ***esp_openthread_get_backbone_netif**(void)

Gets the backbone interface of OpenThread border router.

Returns The backbone interface or NULL if border router not initialized.

void **esp_openthread_register_rcp_failure_handler**(*esp_openthread_rcp_failure_handler* handler)

Registers the callback for RCP failure.

void **esp_openthread_rcp_deinit**(void)

Deinitializes the conneciton to RCP.

Thread is an IPv6-based mesh networking technology for IoT. Code examples for the Thread API are provided in the openthread directory of ESP-IDF examples.

2.5.4 ESP-NETIF

ESP-NETIF

The purpose of ESP-NETIF library is twofold:

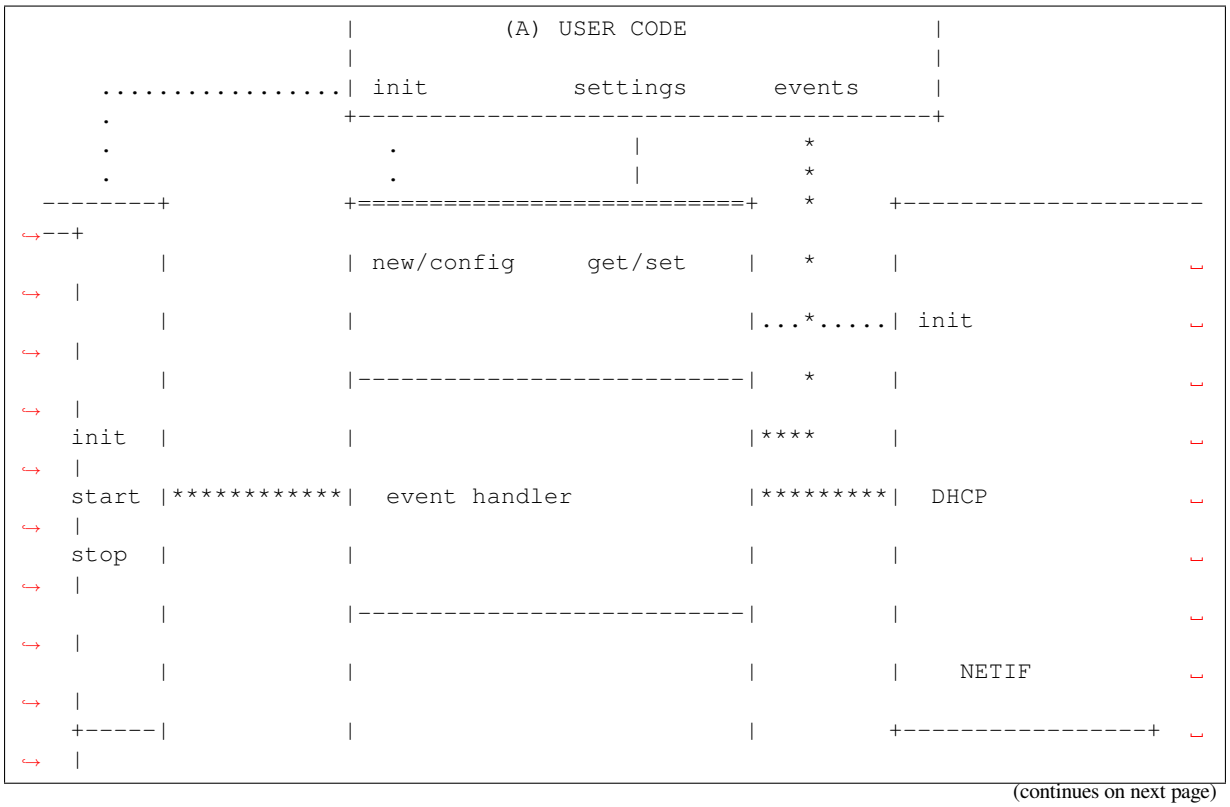
- It provides an abstraction layer for the application on top of the TCP/IP stack. This will allow applications to choose between IP stacks in the future.
- The APIs it provides are thread safe, even if the underlying TCP/IP stack APIs are not.

ESP-IDF currently implements ESP-NETIF for the lwIP TCP/IP stack only. However, the adapter itself is TCP/IP implementation agnostic and different implementations are possible.

Some ESP-NETIF API functions are intended to be called by application code, for example to get/set interface IP addresses, configure DHCP. Other functions are intended for internal ESP-IDF use by the network driver layer.

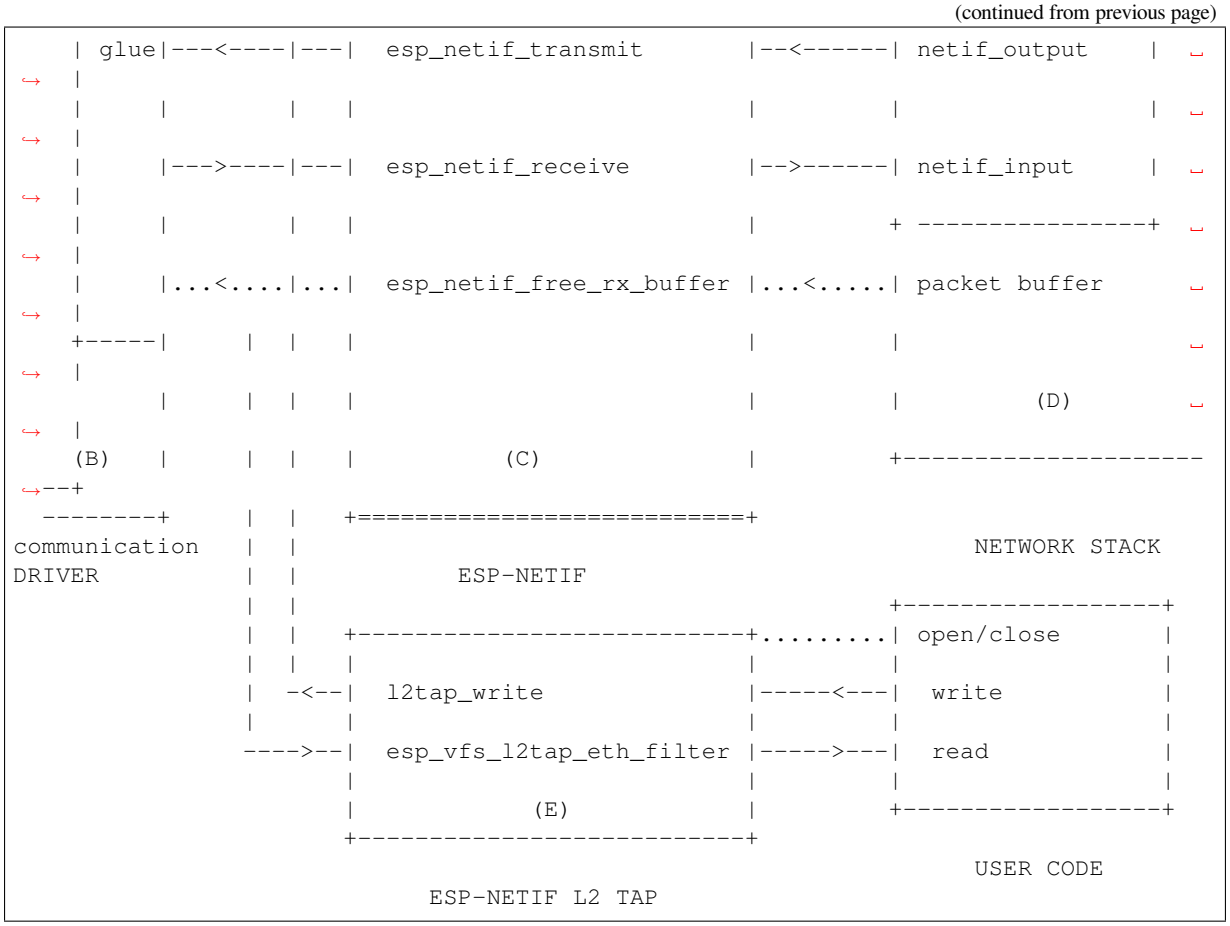
In many cases, applications do not need to call ESP-NETIF APIs directly as they are called from the default network event handlers.

ESP-NETIF architecture



(continues on next page)

(continued from previous page)



Data and event flow in the diagram

- `. . . . . .` Initialization line from user code to ESP-NETIF and communication driver
- `--<--->--` Data packets going from communication media to TCP/IP stack and back
- `********` Events aggregated in ESP-NETIF propagates to driver, user code and network stack
- `|` User settings and runtime configuration

ESP-NETIF interaction

A) User code, boiler plate Overall application interaction with a specific IO driver for communication media and configured TCP/IP network stack is abstracted using ESP-NETIF APIs and outlined as below:

A) Initialization code

1) Initializes IO driver
2) Creates a new instance of ESP-NETIF and configure with
   - ESP-NETIF specific options (flags, behaviour, name)
   - Network stack options (netif init and input functions, not publicly available)
   - IO driver specific options (transmit, free rx buffer functions, IO driver handle)
3) Attaches the IO driver handle to the ESP-NETIF instance created in the above steps
4) Configures event handlers
   - use default handlers for common interfaces defined in IO drivers; or define a specific handlers for customised behaviour/new interfaces
   - register handlers for app related events (such as IP lost/acquired)

B) Interaction with network interfaces using ESP-NETIF API

- Getting and setting TCP/IP related parameters (DHCP, IP, etc)
- Receiving IP events (connect/disconnect)
- Controlling application lifecycle (set interface up/down)

B) Communication driver, IO driver, media driver Communication driver plays these two important roles in relation with ESP-NETIF:

1) Event handlers: Define behaviour patterns of interaction with ESP-NETIF (for example: ethernet link-up -> turn netif on)
2) Glue IO layer: Adapts the input/output functions to use ESP-NETIF transmit, receive and free receive buffer

- Installs driver_transmit to appropriate ESP-NETIF object, so that outgoing packets from network stack are passed to the IO driver
- Calls `esp_netif_receive()` to pass incoming data to network stack

C) ESP-NETIF ESP-NETIF is an intermediary between an IO driver and a network stack, connecting packet data path between these two. As that it provides a set of interfaces for attaching a driver to ESP-NETIF object (runtime) and configuring a network stack (compile time). In addition to that a set of API is provided to control network interface lifecycle and its TCP/IP properties. As an overview, the ESP-NETIF public interface could be divided into these 6 groups:

1) Initialization APIs (to create and configure ESP-NETIF instance)
2) Input/Output API (for passing data between IO driver and network stack)
3) Event or Action API

- Used for network interface lifecycle management
- ESP-NETIF provides building blocks for designing event handlers

4) Setters and Getters for basic network interface properties
5) Network stack abstraction: enabling user interaction with TCP/IP stack

- Set interface up or down
- DHCP server and client API
- DNS API

6) Driver conversion utilities

D) Network stack Network stack has no public interaction with application code with regard to public interfaces and shall be fully abstracted by ESP-NETIF API.

E) ESP-NETIF L2 TAP Interface The ESP-NETIF L2 TAP interface is ESP-IDF mechanism utilized to access Data Link Layer (L2 per OSI/ISO) for frame reception and transmission from user application. Its typical usage in embedded world might be implementation of non-IP related protocols such as PTP, Wake on LAN and others. Note that only Ethernet (IEEE 802.3) is currently supported.

From user perspective, the ESP-NETIF L2 TAP interface is accessed using file descriptors of VFS which provides a file-like interfacing (using functions like `open()`, `read()`, `write()`, etc). Refer to *Virtual filesystem component* to learn more.

There is only one ESP-NETIF L2 TAP interface device (path name) available. However multiple file descriptors with different configuration can be opened at a time since the ESP-NETIF L2 TAP interface can be understood as generic entry point to Layer 2 infrastructure. Important is then specific configuration of particular file descriptor. It can be configured to give an access to specific Network Interface identified by `if_key` (e.g. *ETH_DEF*) and to filter only specific frames based on their type (e.g. Ethernet type in case of IEEE 802.3). Filtering only specific frames is crucial since the ESP-NETIF L2 TAP needs to exist along with IP stack and so the IP related traffic (IP, ARP, etc.) should not be passed directly to the user application. Even though such option is still configurable, it is not recommended in standard use cases. Filtering is also advantageous from a perspective the user's application gets access only to frame types it is interested in and the remaining traffic is either passed to other L2 TAP file descriptors or to IP stack.

ESP-NETIF L2 TAP Interface Usage Manual

Initialization To be able to use the ESP-NETIF L2 TAP interface, it needs to be enabled in Kconfig by *CONFIG_ESP_NETIF_L2_TAP* first and then registered by `esp_vfs_l2tap_intf_register()` prior usage of any VFS function.

open() Once the ESP-NETIF L2 TAP is registered, it can be opened at path name "/dev/net/tap". The same path name can be opened multiple times up to *CONFIG_ESP_NETIF_L2_TAP_MAX_FDS* and multiple file descriptors with with different configuration may access the Data Link Layer frames.

The ESP-NETIF L2 TAP can be opened with `O_NONBLOCK` file status flag to the `read()` does not block. Note that the `write()` may block in current implementation when accessing a Network interface since it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed. The file status flag can be retrieved and modified using `fcntl()`.

On success, `open()` returns the new file descriptor (a nonnegative integer). On error, -1 is returned and `errno` is set to indicate the error.

ioctl() The newly opened ESP-NETIF L2 TAP file descriptor needs to be configured prior its usage since it is not bounded to any specific Network Interface and no frame type filter is configured. The following configuration options are available to do so:

- `L2TAP_S_INTF_DEVICE` - bounds the file descriptor to specific Network Interface which is identified by its `if_key`. ESP-NETIF Network Interface `if_key` is passed to `ioctl()` as the third parameter. Note that default Network Interfaces `if_key`'s used in ESP-IDF can be found in esp_netif/include/esp_netif_defaults.h.
- `L2TAP_S_DEVICE_DRV_HNDL` - is other way how to bound the file descriptor to specific Network Interface. In this case the Network interface is identified directly by IO Driver handle (e.g. `esp_eth_handle_t` in case of Ethernet). The IO Driver handle is passed to `ioctl()` as the third parameter.
- `L2TAP_S_RCV_FILTER` - sets the filter to frames with this type to be passed to the file descriptor. In case of Ethernet frames, the frames are to be filtered based on Length/Ethernet type field. In case the filter value is set less than or equal to 0x05DC, the Ethernet type field is considered to represent IEEE802.3 Length Field and all frames with values in interval <0, 0x05DC> at that field are to be passed to the file descriptor. The IEEE802.2 logical link control (LLC) resolution is then expected to be performed by user's application. In case the filter value is set greater than 0x05DC, the Ethernet type field is considered to represent protocol identification and only frames which are equal to the set value are to be passed to the file descriptor.

All above set configuration options have getter counterpart option to read the current settings.

Warning: The file descriptor needs to be firstly bounded to specific Network Interface by `L2TAP_S_INTF_DEVICE` or `L2TAP_S_DEVICE_DRV_HNDL` to be `L2TAP_S_RCV_FILTER` option available.

Note: VLAN tagged frames are currently not recognized. If user needs to process VLAN tagged frames, they need set filter to be equal to VLAN tag (i.e. 0x8100 or 0x88A8) and process the VLAN tagged frames in user application.

Note: `L2TAP_S_DEVICE_DRV_HNDL` is particularly useful when user's application does not require usage of IP stack and so ESP-NETIF is not required to be initialized too. As a result, Network Interface cannot be identified by its `if_key` and hence it needs to be identified directly by its IO Driver handle.

On success, `ioctl()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.
EBADF - not a valid file descriptor.

EACCES - option change is denied in this state (e.g. file descriptor has not be bounded to Network interface yet).
EINVAL - invalid configuration argument. Ethernet type filter is already used by other file descriptor on that same Network interface.
ENODEV - no such Network Interface which is tried to be assigned to the file descriptor exists.
ENOSYS - unsupported operation, passed configuration option does not exists.

fcntl() `fcntl()` is used to manipulate with properties of opened ESP-NETIF L2 TAP file descriptor.

The following commands manipulate the status flags associated with file descriptor:

- `F_GETFD` - the function returns the file descriptor flags, the third argument is ignored.
- `F_SETFD` - sets the file descriptor flags to the value specified by the third argument. Zero is returned.

On error, -1 is returned, and `errno` is set to indicate the error.
EBADF - not a valid file descriptor.
ENOSYS - unsupported command.

read() Opened and configured ESP-NETIF L2 TAP file descriptor can be accessed by `read()` to get inbound frames. The read operation can be either blocking or non-blocking based on actual state of `O_NONBLOCK` file status flag. When the file status flag is set blocking, the read operation waits until a frame is received and context is switched to other task. When the file status flag is set non-blocking, the read operation returns immediately. In such case, either a frame is returned if it was already queued or the function indicates the queue is empty. The number of queued frames associated with one file descriptor is limited by *CONFIG_ESP_NETIF_L2_TAP_RX_QUEUE_SIZE* Kconfig option. Once the number of queued frames reach configured threshold, the newly arriving frames are dropped until the queue has enough room to accept incoming traffic (Tail Drop queue management).

On success, `read()` returns the number of bytes read. Zero is returned when size of the destination buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.
EBADF - not a valid file descriptor.
EAGAIN - the file descriptor has been marked non-blocking (`O_NONBLOCK`), and the read would block.

write() A raw Data Link Layer frame can be sent to Network Interface via opened and configured ESP-NETIF L2 TAP file descriptor. User's application is responsible to construct the whole frame except for fields which are added automatically by the physical interface device. The following fields need to be constructed by the user's application in case of Ethernet link: source/destination MAC addresses, Ethernet type, actual protocol header and user data. See below for more information about Ethernet frame structure.

```
+-------------------+-------------------+-------------+----------------------------
↪---      --+
|  Destination MAC  |     Source MAC    | Type/Length | Payload (protocol header/
↪data) ...    |
+-------------------+-------------------+-------------+----------------------------
↪---      --+
        6B                   6B                2B                 0-1486B
```

In other words, there is no additional frame processing performed by the ESP-NETIF L2 TAP interface. It only checks the Ethernet type of the frame is the same as the filter configured in the file descriptor. If the Ethernet type is different, an error is returned and the frame is not sent. Note that the `write()` may block in current implementation when accessing a Network interface since it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed.

On success, `write()` returns the number of bytes written. Zero is returned when size of the input buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.

EBADF - not a valid file descriptor.
EBADMSG - Ethernet type of the frame is different then file descriptor configured filter.
EIO - Network interface not available or busy.

close() Opened ESP-NETIF L2 TAP file descriptor can be closed by the `close()` to free its allocated resources. The ESP-NETIF L2 TAP implementation of `close()` may block. On the other hand, it is thread safe and can be called from different task than the file descriptor is actually used. If such situation occurs and one task is blocked in I/O operation and another task tries to close the file descriptor, the first task is unblocked. The first' s task read operation then ends with error.

On success, `close()` returns zero. On error, -1 is returned, and `errno` is set to indicate the error.
EBADF - not a valid file descriptor.

select() Select is used in a standard way, just *CONFIG_VFS_SUPPORT_SELECT* needs to be enabled to be the `select()` function available.

ESP-NETIF programmer' s manual Please refer to the example section for basic initialization of default interfaces:

- WiFi Station: wifi/getting_started/station/main/station_example_main.c
- Ethernet: ethernet/basic/main/ethernet_example_main.c
- L2 TAP: protocols/l2tap/main/l2tap_main.c
- WiFi Access Point: wifi/getting_started/softAP/main/softap_example_main.c

For more specific cases please consult this guide: *ESP-NETIF Custom I/O Driver*.

WiFi default initialization The initialization code as well as registering event handlers for default interfaces, such as softAP and station, are provided in separate APIs to facilitate simple startup code for most applications:

- *esp_netif_create_default_wifi_sta()*
- *esp_netif_create_default_wifi_ap()*

Please note that these functions return the `esp_netif` handle, i.e. a pointer to a network interface object allocated and configured with default settings, which as a consequence, means that:

- The created object has to be destroyed if a network de-initialization is provided by an application using *esp_netif_destroy_default_wifi()*.
- These *default* interfaces must not be created multiple times, unless the created handle is deleted using *esp_netif_destroy()*.
- When using Wifi in `AP+STA` mode, both these interfaces has to be created.

API Reference

Header File

- components/esp_netif/include/esp_netif.h

Functions

esp_err_t **esp_netif_init**(void)

Initialize the underlying TCP/IP stack.

Note: This function should be called exactly once from application code, when the application starts up.

Returns

- ESP_OK on success
- ESP_FAIL if initializing failed

esp_err_t **esp_netif_deinit**(void)

Deinitialize the esp-netif component (and the underlying TCP/IP stack)

```
Note: Deinitialization is not supported yet
```

Returns

- ESP_ERR_INVALID_STATE if esp_netif not initialized
- ESP_ERR_NOT_SUPPORTED otherwise

esp_netif_t ***esp_netif_new**(const *esp_netif_config_t* *esp_netif_config)

Creates an instance of new esp-netif object based on provided config.

Parameters **esp_netif_config** – pointer esp-netif configuration

Returns

- pointer to esp-netif object on success
- NULL otherwise

void **esp_netif_destroy**(*esp_netif_t* *esp_netif)

Destroys the esp_netif object.

Parameters **esp_netif** – **[in]** pointer to the object to be deleted

esp_err_t **esp_netif_set_driver_config**(*esp_netif_t* *esp_netif, const *esp_netif_driver_ifconfig_t* *driver_config)

Configures driver related options of esp_netif object.

Parameters

- **esp_netif** – **[inout]** pointer to the object to be configured
- **driver_config** – **[in]** pointer esp-netif io driver related configuration

Returns

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS if invalid parameters provided

esp_err_t **esp_netif_attach**(*esp_netif_t* *esp_netif, *esp_netif_iodriver_handle* driver_handle)

Attaches esp_netif instance to the io driver handle.

Calling this function enables connecting specific esp_netif object with already initialized io driver to update esp_netif object with driver specific configuration (i.e. calls post_attach callback, which typically sets io driver callbacks to esp_netif instance and starts the driver)

Parameters

- **esp_netif** – **[inout]** pointer to esp_netif object to be attached
- **driver_handle** – **[in]** pointer to the driver handle

Returns

- ESP_OK on success
- ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED if driver's pot_attach callback failed

esp_err_t **esp_netif_receive**(*esp_netif_t* *esp_netif, void *buffer, size_t len, void *eb)

Passes the raw packets from communication media to the appropriate TCP/IP stack.

This function is called from the configured (peripheral) driver layer. The data are then forwarded as frames to the TCP/IP stack.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **buffer** –**[in]** Received data
- **len** –**[in]** Length of the data frame
- **eb** –**[in]** Pointer to internal buffer (used in Wi-Fi driver)

Returns

- ESP_OK

void **esp_netif_action_start**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver start event Creates network interface, if AUTOUP enabled turns the interface on, if DHCPS enabled starts dhcp server.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_stop**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver stop event.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_connected**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver connected event.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_disconnected**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver disconnected event.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_got_ip**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon network got IP event.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_join_ip6_multicast_group**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group join.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_leave_ip6_multicast_group**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group leave.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_add_ip6_address**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 address added by the underlying stack.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** – **[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

void **esp_netif_action_remove_ip6_address**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 address removed by the underlying stack.

Note: This API can be directly used as event handler

Parameters

- **esp_netif** – **[in]** Handle to esp-netif instance
- **base** –
- **event_id** –
- **data** –

esp_err_t **esp_netif_set_default_netif**(*esp_netif_t* *esp_netif)

Manual configuration of the default netif.

This API overrides the automatic configuration of the default interface based on the route_prio If the selected netif is set default using this API, no other interface could be set-default disregarding its route_prio number (unless the selected netif gets destroyed)

Parameters **esp_netif** – **[in]** Handle to esp-netif instance
Returns ESP_OK on success

esp_err_t **esp_netif_join_ip6_multicast_group**(*esp_netif_t* *esp_netif, const *esp_ip6_addr_t* *addr)

Cause the TCP/IP stack to join a IPv6 multicast group.

Parameters

- **esp_netif** – **[in]** Handle to esp-netif instance
- **addr** – **[in]** The multicast group to join

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_MLD6_FAILED
- ESP_ERR_NO_MEM

esp_err_t **esp_netif_leave_ip6_multicast_group**(*esp_netif_t* *esp_netif, const *esp_ip6_addr_t* *addr)

Cause the TCP/IP stack to leave a IPv6 multicast group.

Parameters

- **esp_netif** – **[in]** Handle to esp-netif instance
- **addr** – **[in]** The multicast group to leave

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_MLD6_FAILED
- ESP_ERR_NO_MEM

esp_err_t **esp_netif_set_mac**(*esp_netif_t* *esp_netif, uint8_t mac[])

Set the mac address for the interface instance.

Parameters

- **esp_netif** – **[in]** Handle to esp-netif instance

- **mac** –**[in]** Desired mac address for the related network interface

Returns

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

esp_err_t **esp_netif_get_mac**(*esp_netif_t* *esp_netif, uint8_t mac[])

Get the mac address for the interface instance.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **mac** –**[out]** Resultant mac address for the related network interface

Returns

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

esp_err_t **esp_netif_set_hostname**(*esp_netif_t* *esp_netif, const char *hostname)

Set the hostname of an interface.

The configured hostname overrides the default configuration value CONFIG_LWIP_LOCAL_HOSTNAME. Please note that when the hostname is altered after interface started/connected the changes would only be reflected once the interface restarts/reconnects

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **hostname** –**[in]** New hostname for the interface. Maximum length 32 bytes.

Returns

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

esp_err_t **esp_netif_get_hostname**(*esp_netif_t* *esp_netif, const char **hostname)

Get interface hostname.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **hostname** –**[out]** Returns a pointer to the hostname. May be NULL if no hostname is set. If set non-NULL, pointer remains valid (and string may change if the hostname changes).

Returns

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

bool **esp_netif_is_netif_up**(*esp_netif_t* *esp_netif)

Test if supplied interface is up or down.

Parameters **esp_netif** –**[in]** Handle to esp-netif instance

Returns

- true - Interface is up
- false - Interface is down

esp_err_t **esp_netif_get_ip_info**(*esp_netif_t* *esp_netif, *esp_netif_ip_info_t* *ip_info)

Get interface' s IP address information.

If the interface is up, IP information is read directly from the TCP/IP stack. If the interface is down, IP information is read from a copy kept in the ESP-NETIF instance

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **ip_info** –**[out]** If successful, IP information will be returned in this argument.

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_get_old_ip_info**(*esp_netif_t* *esp_netif, *esp_netif_ip_info_t* *ip_info)

Get interface's old IP information.

Returns an "old" IP address previously stored for the interface when the valid IP changed.

If the IP lost timer has expired (meaning the interface was down for longer than the configured interval) then the old IP information will be zero.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **ip_info** –**[out]** If successful, IP information will be returned in this argument.

Returns
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_set_ip_info**(*esp_netif_t* *esp_netif, const *esp_netif_ip_info_t* *ip_info)

Set interface's IP address information.

This function is mainly used to set a static IP on an interface.

If the interface is up, the new IP information is set directly in the TCP/IP stack.

The copy of IP information kept in the ESP-NETIF instance is also updated (this copy is returned if the IP is queried while the interface is still down.)

Note: DHCP client/server must be stopped (if enabled for this interface) before setting new IP information.

Note: Calling this interface for may generate a SYSTEM_EVENT_STA_GOT_IP or SYSTEM_EVENT_ETH_GOT_IP event.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **ip_info** –**[in]** IP information to set on the specified interface

Returns
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED If DHCP server or client is still running

esp_err_t **esp_netif_set_old_ip_info**(*esp_netif_t* *esp_netif, const *esp_netif_ip_info_t* *ip_info)

Set interface old IP information.

This function is called from the DHCP client (if enabled), before a new IP is set. It is also called from the default handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events.

Calling this function stores the previously configured IP, which can be used to determine if the IP changes in the future.

If the interface is disconnected or down for too long, the "IP lost timer" will expire (after the configured interval) and set the old IP information to zero.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **ip_info** –**[in]** Store the old IP information for the specified interface

Returns
- ESP_OK

- ESP_ERR_ESP_NETIF_INVALID_PARAMS

int **esp_netif_get_netif_impl_index**(*esp_netif_t* *esp_netif)

Get net interface index from network stack implementation.

Note: This index could be used in `setsockopt()` to bind socket with multicast interface

---

Parameters **esp_netif** –**[in]** Handle to esp-netif instance
Returns implementation specific index of interface represented with supplied esp_netif

esp_err_t **esp_netif_get_netif_impl_name**(*esp_netif_t* *esp_netif, char *name)

Get net interface name from network stack implementation.

Note: This name could be used in `setsockopt()` to bind socket with appropriate interface

---

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **name** –**[out]** Interface name as specified in underlying TCP/IP stack. Note that the actual name will be copied to the specified buffer, which must be allocated to hold maximum interface name size (6 characters for lwIP)

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_dhcps_option**(*esp_netif_t* *esp_netif, *esp_netif_dhcp_option_mode_t* opt_op, *esp_netif_dhcp_option_id_t* opt_id, void *opt_val, uint32_t opt_len)

Set or Get DHCP server option.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **opt_op** –**[in]** ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- **opt_id** –**[in]** Option index to get or set, must be one of the supported enum values.
- **opt_val** –**[inout]** Pointer to the option parameter.
- **opt_len** –**[in]** Length of the option parameter.

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

esp_err_t **esp_netif_dhcpc_option**(*esp_netif_t* *esp_netif, *esp_netif_dhcp_option_mode_t* opt_op, *esp_netif_dhcp_option_id_t* opt_id, void *opt_val, uint32_t opt_len)

Set or Get DHCP client option.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **opt_op** –**[in]** ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- **opt_id** –**[in]** Option index to get or set, must be one of the supported enum values.
- **opt_val** –**[inout]** Pointer to the option parameter.
- **opt_len** –**[in]** Length of the option parameter.

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED

- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

esp_err_t **esp_netif_dhcpc_start** (*esp_netif_t* *esp_netif)

Start DHCP client (only if enabled in interface object)

Note: The default event handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events call this function.

Parameters **esp_netif** –**[in]** Handle to esp-netif instance

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED
- ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

esp_err_t **esp_netif_dhcpc_stop** (*esp_netif_t* *esp_netif)

Stop DHCP client (only if enabled in interface object)

Note: Calling action_netif_stop() will also stop the DHCP Client if it is running.

Parameters **esp_netif** –**[in]** Handle to esp-netif instance

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

esp_err_t **esp_netif_dhcpc_get_status** (*esp_netif_t* *esp_netif, *esp_netif_dhcp_status_t* *status)

Get DHCP client status.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **status** –**[out]** If successful, the status of DHCP client will be returned in this argument.

Returns

- ESP_OK

esp_err_t **esp_netif_dhcps_get_status** (*esp_netif_t* *esp_netif, *esp_netif_dhcp_status_t* *status)

Get DHCP Server status.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **status** –**[out]** If successful, the status of the DHCP server will be returned in this argument.

Returns

- ESP_OK

esp_err_t **esp_netif_dhcps_start** (*esp_netif_t* *esp_netif)

Start DHCP server (only if enabled in interface object)

Parameters **esp_netif** –**[in]** Handle to esp-netif instance

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

esp_err_t **esp_netif_dhcps_stop** (*esp_netif_t* *esp_netif)

Stop DHCP server (only if enabled in interface object)

Parameters **esp_netif** –**[in]** Handle to esp-netif instance

Returns

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

esp_err_t **esp_netif_dhcps_get_clients_by_mac**(*esp_netif_t* *esp_netif, int num, *esp_netif_pair_mac_ip_t* *mac_ip_pair)

Populate IP addresses of clients connected to DHCP server listed by their MAC addresses.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **num** –**[in]** Number of clients with specified MAC addresses in the array of pairs
- **mac_ip_pair** –**[inout]** Array of pairs of MAC and IP addresses (MAC are inputs, IP outputs)

Returns

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS on invalid params
- ESP_ERR_NOT_SUPPORTED if DHCP server not enabled

esp_err_t **esp_netif_set_dns_info**(*esp_netif_t* *esp_netif, *esp_netif_dns_type_t* type, *esp_netif_dns_info_t* *dns)

Set DNS Server information.

This function behaves differently if DHCP server or client is enabled

If DHCP client is enabled, main and backup DNS servers will be updated automatically from the DHCP lease if the relevant DHCP options are set. Fallback DNS Server is never updated from the DHCP lease and is designed to be set via this API. If DHCP client is disabled, all DNS server types can be set via this API only.

If DHCP server is enabled, the Main DNS Server setting is used by the DHCP server to provide a DNS Server option to DHCP clients (Wi-Fi stations).

- The default Main DNS server is typically the IP of the DHCP server itself.
- This function can override it by setting server type ESP_NETIF_DNS_MAIN.
- Other DNS Server types are not supported for the DHCP server.
- To propagate the DNS info to client, please stop the DHCP server before using this API.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **type** –**[in]** Type of DNS Server to set: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- **dns** –**[in]** DNS Server address to set

Returns

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

esp_err_t **esp_netif_get_dns_info**(*esp_netif_t* *esp_netif, *esp_netif_dns_type_t* type, *esp_netif_dns_info_t* *dns)

Get DNS Server information.

Return the currently configured DNS Server address for the specified interface and Server type.

This may be result of a previous call to *esp_netif_set_dns_info()*. If the interface' s DHCP client is enabled, the Main or Backup DNS Server may be set by the current DHCP lease.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **type** –**[in]** Type of DNS Server to get: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- **dns** –**[out]** DNS Server result is written here on success

Returns
- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

esp_err_t **esp_netif_create_ip6_linklocal** (*esp_netif_t* *esp_netif)

Create interface link-local IPv6 address.

Cause the TCP/IP stack to create a link-local IPv6 address for the specified interface.

This function also registers a callback for the specified interface, so that if the link-local address becomes verified as the preferred address then a SYSTEM_EVENT_GOT_IP6 event will be sent.

Parameters **esp_netif** –**[in]** Handle to esp-netif instance
Returns
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_get_ip6_linklocal** (*esp_netif_t* *esp_netif, *esp_ip6_addr_t* *if_ip6)

Get interface link-local IPv6 address.

If the specified interface is up and a preferred link-local IPv6 address has been created for the interface, return a copy of it.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **if_ip6** –**[out]** IPv6 information will be returned in this argument if successful.

Returns
- ESP_OK
- ESP_FAIL If interface is down, does not have a link-local IPv6 address, or the link-local IPv6 address is not a preferred address.

esp_err_t **esp_netif_get_ip6_global** (*esp_netif_t* *esp_netif, *esp_ip6_addr_t* *if_ip6)

Get interface global IPv6 address.

If the specified interface is up and a preferred global IPv6 address has been created for the interface, return a copy of it.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **if_ip6** –**[out]** IPv6 information will be returned in this argument if successful.

Returns
- ESP_OK
- ESP_FAIL If interface is down, does not have a global IPv6 address, or the global IPv6 address is not a preferred address.

int **esp_netif_get_all_ip6** (*esp_netif_t* *esp_netif, *esp_ip6_addr_t* if_ip6[])

Get all IPv6 addresses of the specified interface.

Parameters
- **esp_netif** –**[in]** Handle to esp-netif instance
- **if_ip6** –**[out]** Array of IPv6 addresses will be copied to the argument

Returns number of returned IPv6 addresses

void **esp_netif_set_ip4_addr** (*esp_ip4_addr_t* *addr, uint8_t a, uint8_t b, uint8_t c, uint8_t d)

Sets IPv4 address to the specified octets.

Parameters
- **addr** –**[out]** IP address to be set
- **a** –the first octet (127 for IP 127.0.0.1)
- **b** –
- **c** –
- **d** –

char *`esp_ip4addr_ntoa` (const *esp_ip4_addr_t* *addr, char *buf, int buflen)

Converts numeric IP address into decimal dotted ASCII representation.

Parameters
- **`addr`** –ip address in network order to convert
- **`buf`** –target buffer where the string is stored
- **`buflen`** –length of buf

Returns either pointer to buf which now holds the ASCII representation of addr or NULL if buf was too small

uint32_t `esp_ip4addr_aton` (const char *addr)

Ascii internet address interpretation routine The value returned is in network order.

Parameters **`addr`** –IP address in ascii representation (e.g. "127.0.0.1")
Returns ip address in network order

esp_err_t `esp_netif_str_to_ip4` (const char *src, *esp_ip4_addr_t* *dst)

Converts Ascii internet IPv4 address into esp_ip4_addr_t.

Parameters
- **`src`** –**[in]** IPv4 address in ascii representation (e.g. "127.0.0.1")
- **`dst`** –**[out]** Address of the target esp_ip4_addr_t structure to receive converted address

Returns
- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into

esp_err_t `esp_netif_str_to_ip6` (const char *src, *esp_ip6_addr_t* *dst)

Converts Ascii internet IPv6 address into esp_ip4_addr_t Zeros in the IP address can be stripped or completely ommited: "2001:db8:85a3:0:0:0:2:1" or "2001:db8::2:1")

Parameters
- **`src`** –**[in]** IPv6 address in ascii representation (e.g. "2001:0db8:85a3:0000:0000:0000:0002:0001")
- **`dst`** –**[out]** Address of the target esp_ip6_addr_t structure to receive converted address

Returns
- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into

esp_netif_iodriver_handle `esp_netif_get_io_driver` (*esp_netif_t* *esp_netif)

Gets media driver handle for this esp-netif instance.

Parameters **`esp_netif`** –**[in]** Handle to esp-netif instance
Returns opaque pointer of related IO driver

esp_netif_t *`esp_netif_get_handle_from_ifkey` (const char *if_key)

Searches over a list of created objects to find an instance with supplied if key.

Parameters **`if_key`** –Textual description of network interface
Returns Handle to esp-netif instance

esp_netif_flags_t `esp_netif_get_flags` (*esp_netif_t* *esp_netif)

Returns configured flags for this interface.

Parameters **`esp_netif`** –**[in]** Handle to esp-netif instance
Returns Configuration flags

const char *`esp_netif_get_ifkey` (*esp_netif_t* *esp_netif)

Returns configured interface key for this esp-netif instance.

Parameters **`esp_netif`** –**[in]** Handle to esp-netif instance
Returns Textual description of related interface

const char *`esp_netif_get_desc`(*esp_netif_t* *esp_netif)

Returns configured interface type for this esp-netif instance.

Parameters `esp_netif` –**[in]** Handle to esp-netif instance
Returns Enumerated type of this interface, such as station, AP, ethernet

int `esp_netif_get_route_prio`(*esp_netif_t* *esp_netif)

Returns configured routing priority number.

Parameters `esp_netif` –**[in]** Handle to esp-netif instance
Returns Integer representing the instance's route-prio, or -1 if invalid paramters

int32_t `esp_netif_get_event_id`(*esp_netif_t* *esp_netif, *esp_netif_ip_event_type_t* event_type)

Returns configured event for this esp-netif instance and supplied event type.

Parameters
- `esp_netif` –**[in]** Handle to esp-netif instance
- `event_type` –(either get or lost IP)

Returns specific event id which is configured to be raised if the interface lost or acquired IP address
-1 if supplied event_type is not known

esp_netif_t *`esp_netif_next`(*esp_netif_t* *esp_netif)

Iterates over list of interfaces. Returns first netif if NULL given as parameter.

You can use *esp_netif_next_unsafe()* directly if all the system interfaces are under your control and you can safely iterate over them. Otherwise, iterate over interfaces using *esp_netif_tcpip_exec()*, or use *esp_netif_find_if()* to search in the list of netifs with defined predicate.

Note: This API doesn't lock the list, nor the TCPIP context, as this it's usually required to get atomic access between iteration steps rather that within a single iteration. Therefore it is recommended to iterate over the interfaces inside *esp_netif_tcpip_exec()*

Parameters `esp_netif` –**[in]** Handle to esp-netif instance
Returns First netif from the list if supplied parameter is NULL, next one otherwise

esp_netif_t *`esp_netif_next_unsafe`(*esp_netif_t* *esp_netif)

Iterates over list of interfaces without list locking. Returns first netif if NULL given as parameter.

Used for bulk search loops within TCPIP context, e.g. using *esp_netif_tcpip_exec()*, or if we're sure that the iteration is safe from our application perspective (e.g. no interface is removed between iterations)

Parameters `esp_netif` –**[in]** Handle to esp-netif instance
Returns First netif from the list if supplied parameter is NULL, next one otherwise

esp_netif_t *`esp_netif_find_if`(*esp_netif_find_predicate_t* fn, void *ctx)

Return a netif pointer for the first interface that meets criteria defined by the callback.

Parameters
- `fn` –Predicate function returning true for the desired interface
- `ctx` –Context pointer passed to the predicate, typically a descriptor to compare with

Returns valid netif pointer if found, NULL if not

size_t `esp_netif_get_nr_of_ifs`(void)

Returns number of registered esp_netif objects.

Returns Number of esp_netifs

void `esp_netif_netstack_buf_ref`(void *netstack_buf)

increase the reference counter of net stack buffer

Parameters `netstack_buf` –**[in]** the net stack buffer

void **esp_netif_netstack_buf_free**(void *netstack_buf)

free the netstack buffer

Parameters **netstack_buf** –**[in]** the net stack buffer

esp_err_t **esp_netif_tcpip_exec**(*esp_netif_callback_fn* fn, void *ctx)

Utility to execute the supplied callback in TCP/IP context.

Parameters

- **fn** –Pointer to the callback
- **ctx** –Parameter to the callback

Returns The error code (esp_err_t) returned by the callback

Type Definitions

typedef bool (***esp_netif_find_predicate_t**)(*esp_netif_t* *netif, void *ctx)

Predicate callback for *esp_netif_find_if()* used to find interface which meets defined criteria.

typedef *esp_err_t* (***esp_netif_callback_fn**)(void *ctx)

TCPIP thread safe callback used with *esp_netif_tcpip_exec()*

Header File

- components/esp_netif/include/esp_netif_types.h

Structures

struct **esp_netif_dns_info_t**

DNS server info.

Public Members

esp_ip_addr_t **ip**

IPV4 address of DNS server

struct **esp_netif_ip_info_t**

Event structure for IP_EVENT_STA_GOT_IP, IP_EVENT_ETH_GOT_IP events

Public Members

esp_ip4_addr_t **ip**

Interface IPV4 address

esp_ip4_addr_t **netmask**

Interface IPV4 netmask

esp_ip4_addr_t **gw**

Interface IPV4 gateway address

struct **esp_netif_ip6_info_t**

IPV6 IP address information.

Public Members

esp_ip6_addr_t **ip**

Interface IPV6 address

struct **ip_event_got_ip_t**

Event structure for IP_EVENT_GOT_IP event.

Public Members

esp_netif_t ***esp_netif**

Pointer to corresponding esp-netif object

esp_netif_ip_info_t **ip_info**

IP address, netmask, gatway IP address

bool **ip_changed**

Whether the assigned IP has changed or not

struct **ip_event_got_ip6_t**

Event structure for IP_EVENT_GOT_IP6 event

Public Members

esp_netif_t ***esp_netif**

Pointer to corresponding esp-netif object

esp_netif_ip6_info_t **ip6_info**

IPv6 address of the interface

int **ip_index**

IPv6 address index

struct **ip_event_add_ip6_t**

Event structure for ADD_IP6 event

Public Members

esp_ip6_addr_t **addr**

The address to be added to the interface

bool **preferred**

The default preference of the address

struct **ip_event_ap_staipassigned_t**

Event structure for IP_EVENT_AP_STAIPASSIGNED event

Public Members

esp_netif_t ***esp_netif**

Pointer to the associated netif handle

esp_ip4_addr_t **ip**

IP address which was assigned to the station

uint8_t **mac**[6]

MAC address of the connected client

struct **bridgeif_config**

LwIP bridge configuration

Public Members

uint16_t **max_fdb_dyn_entries**

maximum number of entries in dynamic forwarding database

uint16_t **max_fdb_sta_entries**

maximum number of entries in static forwarding database

uint8_t **max_ports**

maximum number of ports the bridge can consist of

struct **esp_netif_inherent_config**

ESP-netif inherent config parameters.

Public Members

esp_netif_flags_t **flags**

flags that define esp-netif behavior

uint8_t **mac**[6]

initial mac address for this interface

const *esp_netif_ip_info_t* ***ip_info**

initial ip address for this interface

uint32_t **get_ip_event**

event id to be raised when interface gets an IP

uint32_t **lost_ip_event**

event id to be raised when interface losts its IP

const char ***if_key**

string identifier of the interface

const char ***if_desc**

textual description of the interface

int **route_prio**

numeric priority of this interface to become a default routing if (if other netifs are up). A higher value of route_prio indicates a higher priority

bridgeif_config_t ***bridge_info**

LwIP bridge configuration

struct **esp_netif_driver_base_s**

ESP-netif driver base handle.

Public Members

esp_err_t (***post_attach**)(*esp_netif_t* *netif, *esp_netif_iodriver_handle* h)

post attach function pointer

esp_netif_t ***netif**

netif handle

struct **esp_netif_driver_ifconfig**

Specific IO driver configuration.

Public Members

esp_netif_iodriver_handle **handle**

io-driver handle

esp_err_t (***transmit**)(void *h, void *buffer, size_t len)

transmit function pointer

esp_err_t (***transmit_wrap**)(void *h, void *buffer, size_t len, void *netstack_buffer)

transmit wrap function pointer

void (***driver_free_rx_buffer**)(void *h, void *buffer)

free rx buffer function pointer

struct **esp_netif_config**

Generic esp_netif configuration.

Public Members

const *esp_netif_inherent_config_t* ***base**

base config

const *esp_netif_driver_ifconfig_t* ***driver**

driver config

const *esp_netif_netstack_config_t* ***stack**

stack config

struct **esp_netif_pair_mac_ip_t**

DHCP client’ s addr info (pair of MAC and IP address)

Public Members

uint8_t **mac**[6]

Clients MAC address

esp_ip4_addr_t **ip**

Clients IP address

Macros

ESP_ERR_ESP_NETIF_BASE

Definition of ESP-NETIF based errors.

ESP_ERR_ESP_NETIF_INVALID_PARAMS

ESP_ERR_ESP_NETIF_IF_NOT_READY

ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED

ESP_ERR_ESP_NETIF_NO_MEM

ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED

ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED

ESP_ERR_ESP_NETIF_INIT_FAILED

ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED

ESP_ERR_ESP_NETIF_MLD6_FAILED

ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED

ESP_ERR_ESP_NETIF_DHCPS_START_FAILED

ESP_NETIF_BR_FLOOD

Definition of ESP-NETIF bridge controll.

ESP_NETIF_BR_DROP

ESP_NETIF_BR_FDW_CPU

Type Definitions

typedef struct esp_netif_obj **esp_netif_t**

typedef enum *esp_netif_flags* **esp_netif_flags_t**

typedef enum *esp_netif_ip_event_type* **esp_netif_ip_event_type_t**

typedef struct *bridgeif_config* **bridgeif_config_t**

LwIP bridge configuration

typedef struct *esp_netif_inherent_config* **esp_netif_inherent_config_t**

ESP-netif inherent config parameters.

typedef struct *esp_netif_config* **esp_netif_config_t**

typedef void ***esp_netif_iodriver_handle**

IO driver handle type.

typedef struct *esp_netif_driver_base_s* **esp_netif_driver_base_t**

ESP-netif driver base handle.

typedef struct *esp_netif_driver_ifconfig* **esp_netif_driver_ifconfig_t**

typedef struct esp_netif_netstack_config **esp_netif_netstack_config_t**

Specific L3 network stack configuration.

typedef *esp_err_t* (***esp_netif_receive_t**)(*esp_netif_t* *esp_netif, void *buffer, size_t len, void *eb)

ESP-NETIF Receive function type.

Enumerations

enum **esp_netif_dns_type_t**

Type of DNS server.

Values:

enumerator **ESP_NETIF_DNS_MAIN**

DNS main server address

enumerator **ESP_NETIF_DNS_BACKUP**

DNS backup server address (Wi-Fi STA and Ethernet only)

enumerator **ESP_NETIF_DNS_FALLBACK**

DNS fallback server address (Wi-Fi STA and Ethernet only)

enumerator **ESP_NETIF_DNS_MAX**

enum **esp_netif_dhcp_status_t**

Status of DHCP client or DHCP server.

Values:

enumerator **ESP_NETIF_DHCP_INIT**

DHCP client/server is in initial state (not yet started)

enumerator **ESP_NETIF_DHCP_STARTED**

DHCP client/server has been started

enumerator **ESP_NETIF_DHCP_STOPPED**

DHCP client/server has been stopped

enumerator **ESP_NETIF_DHCP_STATUS_MAX**

enum **esp_netif_dhcp_option_mode_t**

Mode for DHCP client or DHCP server option functions.

Values:

enumerator **ESP_NETIF_OP_START**

enumerator **ESP_NETIF_OP_SET**

Set option

enumerator **ESP_NETIF_OP_GET**

Get option

enumerator **ESP_NETIF_OP_MAX**

enum **esp_netif_dhcp_option_id_t**

Supported options for DHCP client or DHCP server.

Values:

enumerator **ESP_NETIF_SUBNET_MASK**

Network mask

enumerator **ESP_NETIF_DOMAIN_NAME_SERVER**

Domain name server

enumerator **ESP_NETIF_ROUTER_SOLICITATION_ADDRESS**

Solicitation router address

enumerator **ESP_NETIF_REQUESTED_IP_ADDRESS**

Request specific IP address

enumerator **ESP_NETIF_IP_ADDRESS_LEASE_TIME**

Request IP address lease time

enumerator **ESP_NETIF_IP_REQUEST_RETRY_TIME**

Request IP address retry counter

enumerator **ESP_NETIF_VENDOR_CLASS_IDENTIFIER**

Vendor Class Identifier of a DHCP client

enumerator **ESP_NETIF_VENDOR_SPECIFIC_INFO**

Vendor Specific Information of a DHCP server

enum **ip_event_t**

IP event declarations

Values:

enumerator **IP_EVENT_STA_GOT_IP**

station got IP from connected AP

enumerator **IP_EVENT_STA_LOST_IP**

station lost IP and the IP is reset to 0

enumerator **IP_EVENT_AP_STAIPASSIGNED**

soft-AP assign an IP to a connected station

enumerator **IP_EVENT_GOT_IP6**

station or ap or ethernet interface v6IP addr is preferred

enumerator **IP_EVENT_ETH_GOT_IP**

ethernet got IP from connected AP

enumerator **IP_EVENT_ETH_LOST_IP**

ethernet lost IP and the IP is reset to 0

enumerator **IP_EVENT_PPP_GOT_IP**

PPP interface got IP

enumerator **IP_EVENT_PPP_LOST_IP**

PPP interface lost IP

enum **esp_netif_flags**

Values:

enumerator **ESP_NETIF_DHCP_CLIENT**

enumerator **ESP_NETIF_DHCP_SERVER**

enumerator **ESP_NETIF_FLAG_AUTOUP**

enumerator **ESP_NETIF_FLAG_GARP**

enumerator **ESP_NETIF_FLAG_EVENT_IP_MODIFIED**

enumerator **ESP_NETIF_FLAG_IS_PPP**

enumerator **ESP_NETIF_FLAG_IS_BRIDGE**

enumerator **ESP_NETIF_FLAG_MLDV6_REPORT**

enum **esp_netif_ip_event_type**

Values:

enumerator **ESP_NETIF_IP_EVENT_GOT_IP**

enumerator **ESP_NETIF_IP_EVENT_LOST_IP**

Header File

- components/esp_netif/include/esp_netif_ip_addr.h

Functions

esp_ip6_addr_type_t **esp_netif_ip6_get_addr_type**(*esp_ip6_addr_t* *ip6_addr)

Get the IPv6 address type.

Parameters **ip6_addr** –**[in]** IPv6 type
Returns IPv6 type in form of enum esp_ip6_addr_type_t

static inline void **esp_netif_ip_addr_copy**(*esp_ip_addr_t* *dest, const *esp_ip_addr_t* *src)

Copy IP addresses.

Parameters
- **dest** –**[out]** destination IP
- **src** –**[in]** source IP

Structures

struct **esp_ip6_addr**

IPv6 address.

Public Members

uint32_t **addr**[4]

IPv6 address

uint8_t **zone**

zone ID

struct **esp_ip4_addr**

IPv4 address.

Public Members

uint32_t **addr**

IPv4 address

struct **_ip_addr**

IP address.

Public Members

esp_ip6_addr_t **ip6**

IPv6 address type

esp_ip4_addr_t **ip4**

IPv4 address type

union *_ip_addr*::[anonymous] **u_addr**

IP address union

uint8_t **type**

ipaddress type

Macros

esp_netif_htonl(x)

esp_netif_ip4_makeu32(a, b, c, d)

ESP_IP6_ADDR_BLOCK1(ip6addr)

ESP_IP6_ADDR_BLOCK2(ip6addr)

ESP_IP6_ADDR_BLOCK3(ip6addr)

ESP_IP6_ADDR_BLOCK4(ip6addr)

ESP_IP6_ADDR_BLOCK5(ip6addr)

ESP_IP6_ADDR_BLOCK6(ip6addr)

ESP_IP6_ADDR_BLOCK7(ip6addr)

ESP_IP6_ADDR_BLOCK8(ip6addr)

IPSTR

esp_ip4_addr_get_byte(ipaddr, idx)

esp_ip4_addr1 (ipaddr)

esp_ip4_addr2 (ipaddr)

esp_ip4_addr3 (ipaddr)

esp_ip4_addr4 (ipaddr)

esp_ip4_addr1_16 (ipaddr)

esp_ip4_addr2_16 (ipaddr)

esp_ip4_addr3_16 (ipaddr)

esp_ip4_addr4_16 (ipaddr)

IP2STR (ipaddr)

IPV6STR

IPV62STR (ipaddr)

ESP_IPADDR_TYPE_V4

ESP_IPADDR_TYPE_V6

ESP_IPADDR_TYPE_ANY

ESP_IP4TOUINT32 (a, b, c, d)

ESP_IP4TOADDR (a, b, c, d)

ESP_IP4ADDR_INIT (a, b, c, d)

ESP_IP6ADDR_INIT (a, b, c, d)

Type Definitions

typedef struct *esp_ip4_addr* **esp_ip4_addr_t**

typedef struct *esp_ip6_addr* **esp_ip6_addr_t**

typedef struct *_ip_addr* **esp_ip_addr_t**

IP address.

Enumerations

enum **esp_ip6_addr_type_t**

Values:

enumerator **ESP_IP6_ADDR_IS_UNKNOWN**

enumerator **ESP_IP6_ADDR_IS_GLOBAL**

enumerator **ESP_IP6_ADDR_IS_LINK_LOCAL**

enumerator **ESP_IP6_ADDR_IS_SITE_LOCAL**

enumerator **ESP_IP6_ADDR_IS_UNIQUE_LOCAL**

enumerator **ESP_IP6_ADDR_IS_IPV4_MAPPED_IPV6**

Header File

- components/esp_netif/include/esp_vfs_l2tap.h

Functions

esp_err_t **esp_vfs_l2tap_intf_register**(*l2tap_vfs_config_t* *config)

Add L2 TAP virtual filesystem driver.

This function must be called prior usage of ESP-NETIF L2 TAP Interface

Parameters **config** –L2 TAP virtual filesystem driver configuration. Default base path /dev/net/tap is used when this paramenter is NULL.

Returns esp_err_t

- ESP_OK on success

esp_err_t **esp_vfs_l2tap_intf_unregister**(const char *base_path)

Removes L2 TAP virtual filesystem driver.

Parameters **base_path** –Base path to the L2 TAP virtual filesystem driver. Default path /dev/net/tap is used when this paramenter is NULL.

Returns esp_err_t

- ESP_OK on success

esp_err_t **esp_vfs_l2tap_eth_filter**(*l2tap_iodriver_handle* driver_handle, void *buff, size_t *size)

Filters received Ethernet L2 frames into L2 TAP infrastructure.

Parameters

- **driver_handle** –handle of driver at which the frame was received
- **buff** –received L2 frame
- **size** –input length of the L2 frame which is set to 0 when frame is filtered into L2 TAP

Returns esp_err_t

- ESP_OK is always returned

Structures

struct **l2tap_vfs_config_t**

L2Tap VFS config parameters.

Public Members

const char ***base_path**

vfs base path

Macros

L2TAP_VFS_DEFAULT_PATH

L2TAP_VFS_CONFIG_DEFAULT()

Type Definitions

typedef void ***l2tap_iodriver_handle**

Enumerations

enum **l2tap_ioctl_opt_t**

Values:

enumerator **L2TAP_S_RCV_FILTER**

enumerator **L2TAP_G_RCV_FILTER**

enumerator **L2TAP_S_INTF_DEVICE**

enumerator **L2TAP_G_INTF_DEVICE**

enumerator **L2TAP_S_DEVICE_DRV_HNDL**

enumerator **L2TAP_G_DEVICE_DRV_HNDL**

WiFi default API reference

Header File

- components/esp_wifi/include/esp_wifi_default.h

Functions

esp_err_t **esp_netif_attach_wifi_station**(*esp_netif_t* *esp_netif)

Attaches wifi station interface to supplied netif.

Parameters **esp_netif** – instance to attach the wifi station to

Returns

- ESP_OK on success
- ESP_FAIL if attach failed

esp_err_t **esp_netif_attach_wifi_ap**(*esp_netif_t* *esp_netif)

Attaches wifi soft AP interface to supplied netif.

Parameters **esp_netif** – instance to attach the wifi AP to

Returns

- ESP_OK on success
- ESP_FAIL if attach failed

esp_err_t **esp_wifi_set_default_wifi_sta_handlers**(void)

Sets default wifi event handlers for STA interface.

Returns

- ESP_OK on success, error returned from esp_event_handler_register if failed

esp_err_t **esp_wifi_set_default_wifi_ap_handlers**(void)

Sets default wifi event handlers for AP interface.

Returns

- ESP_OK on success, error returned from esp_event_handler_register if failed

esp_err_t **esp_wifi_clear_default_wifi_driver_and_handlers** (void *esp_netif)

Clears default wifi event handlers for supplied network interface.

Parameters **esp_netif** –instance of corresponding if object

Returns

- ESP_OK on success, error returned from esp_event_handler_register if failed

esp_netif_t ***esp_netif_create_default_wifi_ap** (void)

Creates default WIFI AP. In case of any init error this API aborts.

Note: The API creates esp_netif object with default WiFi access point config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

Returns pointer to esp-netif instance

esp_netif_t ***esp_netif_create_default_wifi_sta** (void)

Creates default WIFI STA. In case of any init error this API aborts.

Note: The API creates esp_netif object with default WiFi station config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

Returns pointer to esp-netif instance

void **esp_netif_destroy_default_wifi** (void *esp_netif)

Destroys default WIFI netif created with esp_netif_create_default_wifi_···() API.

Note: This API unregisters wifi handlers and detaches the created object from the wifi. (this function is a no-operation if esp_netif is NULL)

Parameters **esp_netif** –**[in]** object to detach from WiFi and destroy

esp_netif_t ***esp_netif_create_wifi** (*wifi_interface_t* wifi_if, *esp_netif_inherent_config_t* *esp_netif_config)

Creates esp_netif WiFi object based on the custom configuration.

Attention This API DOES NOT register default handlers!

Parameters

- **wifi_if** –**[in]** type of wifi interface
- **esp_netif_config** –inherent esp-netif configuration pointer

Returns pointer to esp-netif instance

esp_err_t **esp_netif_create_default_wifi_mesh_netifs** (*esp_netif_t* **p_netif_sta, *esp_netif_t* **p_netif_ap)

Creates default STA and AP network interfaces for esp-mesh.

Both netifs are almost identical to the default station and softAP, but with DHCP client and server disabled. Please note that the DHCP client is typically enabled only if the device is promoted to a root node.

Returns created interfaces which could be ignored setting parameters to NULL if an application code does not need to save the interface instances for further processing.

Parameters
- **p_netif_sta** –**[out]** pointer where the resultant STA interface is saved (if non NULL)
- **p_netif_ap** –**[out]** pointer where the resultant AP interface is saved (if non NULL)

Returns ESP_OK on success

2.5.5 IP Network Layer

ESP-NETIF Custom I/O Driver

This section outlines implementing a new I/O driver with esp-netif connection capabilities. By convention the I/O driver has to register itself as an esp-netif driver and thus holds a dependency on esp-netif component and is responsible for providing data path functions, post-attach callback and in most cases also default event handlers to define network interface actions based on driver's lifecycle transitions.

Packet input/output As shown in the diagram, the following three API functions for the packet data path must be defined for connecting with esp-netif:

- *esp_netif_transmit()*
- *esp_netif_free_rx_buffer()*
- *esp_netif_receive()*

The first two functions for transmitting and freeing the rx buffer are provided as callbacks, i.e. they get called from esp-netif (and its underlying TCP/IP stack) and I/O driver provides their implementation.

The receiving function on the other hand gets called from the I/O driver, so that the driver's code simply calls *esp_netif_receive()* on a new data received event.

Post attach callback A final part of the network interface initialization consists of attaching the esp-netif instance to the I/O driver, by means of calling the following API:

```
esp_err_t esp_netif_attach(esp_netif_t *esp_netif, esp_netif_iodriver_handle␣
↪driver_handle);
```

It is assumed that the `esp_netif_iodriver_handle` is a pointer to driver's object, a struct derived from `struct esp_netif_driver_base_s`, so that the first member of I/O driver structure must be this base structure with pointers to

- post-attach function callback
- related esp-netif instance

As a consequence the I/O driver has to create an instance of the struct per below:

```
typedef struct my_netif_driver_s {
        esp_netif_driver_base_t base;           /*!< base structure reserved as␣
↪esp-netif driver */
        driver_impl             *h;             /*!< handle of driver␣
↪implementation */
    } my_netif_driver_t;
```

with actual values of `my_netif_driver_t::base.post_attach` and the actual drivers handle `my_netif_driver_t::h`. So when the *esp_netif_attach()* gets called from the initialization code, the post-attach callback from I/O driver's code gets executed to mutually register callbacks between esp-netif and I/O driver instances. Typically the driver is started as well in the post-attach callback. An example of a simple post-attach callback is outlined below:

```
static esp_err_t my_post_attach_start(esp_netif_t * esp_netif, void * args)
{
    my_netif_driver_t *driver = args;
    const esp_netif_driver_ifconfig_t driver_ifconfig = {
```

(continues on next page)

(continued from previous page)

```
            .driver_free_rx_buffer = my_free_rx_buf,
            .transmit = my_transmit,
            .handle = driver->driver_impl
    };
    driver->base.netif = esp_netif;
    ESP_ERROR_CHECK(esp_netif_set_driver_config(esp_netif, &driver_ifconfig));
    my_driver_start(driver->driver_impl);
    return ESP_OK;
}
```

Default handlers I/O drivers also typically provide default definitions of lifecycle behaviour of related network interfaces based on state transitions of I/O drivers. For example *driver start -> network start*, etc. An example of such a default handler is provided below:

```
esp_err_t my_driver_netif_set_default_handlers(my_netif_driver_t *driver, esp_
↪netif_t * esp_netif)
{
    driver_set_event_handler(driver->driver_impl, esp_netif_action_start, MY_DRV_
↪EVENT_START, esp_netif);
    driver_set_event_handler(driver->driver_impl, esp_netif_action_stop, MY_DRV_
↪EVENT_STOP, esp_netif);
    return ESP_OK;
}
```

Network stack connection The packet data path functions for transmitting and freeing the rx buffer (defined in the I/O driver) are called from the esp-netif, specifically from its TCP/IP stack connecting layer.

Note, that IDF provides several network stack configurations for the most common network interfaces, such as for the WiFi station or Ethernet. These configurations are defined in esp_netif/include/esp_netif_defaults.h and should be sufficient for most network drivers. (In rare cases, expert users might want to define custom lwIP based interface layers; it is possible, but an explicit dependency to lwIP needs to be set)

The following API reference outlines these network stack interaction with the esp-netif:

Header File

- components/esp_netif/include/esp_netif_net_stack.h

Functions

esp_netif_t ***esp_netif_get_handle_from_netif_impl*** (void *dev)

Returns esp-netif handle.

Parameters **dev** –**[in]** opaque ptr to network interface of specific TCP/IP stack
Returns handle to related esp-netif instance

void ***esp_netif_get_netif_impl*** (*esp_netif_t* *esp_netif)

Returns network stack specific implementation handle (if supported)

Note that it is not supported to acquire PPP netif impl pointer and this function will return NULL for esp_netif instances configured to PPP mode

Parameters **esp_netif** –**[in]** Handle to esp-netif instance
Returns handle to related network stack netif handle

esp_err_t **esp_netif_set_link_speed** (*esp_netif_t* *esp_netif, uint32_t speed)

Set link-speed for the specified network interface.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance

- **speed** –**[in]** Link speed in bit/s

Returns ESP_OK on success

esp_err_t **esp_netif_transmit**(*esp_netif_t* *esp_netif, void *data, size_t len)

Outputs packets from the TCP/IP stack to the media to be transmitted.

This function gets called from network stack to output packets to IO driver.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **data** –**[in]** Data to be transmitted
- **len** –**[in]** Length of the data frame

Returns ESP_OK on success, an error passed from the I/O driver otherwise

esp_err_t **esp_netif_transmit_wrap**(*esp_netif_t* *esp_netif, void *data, size_t len, void *netstack_buf)

Outputs packets from the TCP/IP stack to the media to be transmitted.

This function gets called from network stack to output packets to IO driver.

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **data** –**[in]** Data to be transmitted
- **len** –**[in]** Length of the data frame
- **netstack_buf** –**[in]** net stack buffer

Returns ESP_OK on success, an error passed from the I/O driver otherwise

void **esp_netif_free_rx_buffer**(void *esp_netif, void *buffer)

Free the rx buffer allocated by the media driver.

This function gets called from network stack when the rx buffer to be freed in IO driver context, i.e. to deallocate a buffer owned by io driver (when data packets were passed to higher levels to avoid copying)

Parameters

- **esp_netif** –**[in]** Handle to esp-netif instance
- **buffer** –**[in]** Rx buffer pointer

Code examples for TCP/IP socket APIs are provided in the protocols/sockets directory of ESP-IDF examples.

2.5.6 Application Layer

Documentation for Application layer network protocols (above the IP Network layer) are provided in *Application Protocols*.

2.6 Peripherals API

2.6.1 Analog to Digital Converter (ADC) Oneshot Mode Driver

Introduction

The Analog to Digital Converter is an on-chip sensor which is able to measure analog signals from dedicated analog IO pads.

The ADC on ESP32 can be used in scenario(s) like:

- Generate one-shot ADC conversion result
- Generate continuous ADC conversion results

This guide will introduce ADC oneshot mode conversion.

Functional Overview

The following sections of this document cover the typical steps to install and operate an ADC:

- *Resource Allocation* - covers which parameters should be set up to get an ADC handle and how to recycle the resources when ADC finishes working.
- *Unit Configuration* - covers the parameters that should be set up to configure the ADC unit, so as to get ADC conversion raw result.
- *Read Conversion Result* - covers how to get ADC conversion raw result.
- *Hardware Limitations* - describes the ADC related hardware limitations.
- *Power Management* - covers power management related.
- *IRAM Safe* - describes tips on how to read ADC conversion raw result when cache is disabled.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.
- *Kconfig Options* - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

Resource Allocation The ADC oneshot mode driver is implemented based on ESP32 SAR ADC module. Different ESP chips might have different number of independent ADCs. From oneshot mode driver' s point of view, an ADC instance is represented by `adc_oneshot_unit_handle_t`.

To install an ADC instance, set up the required initial configuration structure `adc_oneshot_unit_init_cfg_t`:

- `adc_oneshot_unit_init_cfg_t::unit_id` selects the ADC. Please refer to the datasheet to know dedicated analog IOs for this ADC.
- `adc_oneshot_unit_init_cfg_t::ulp_mode` sets if the ADC will be working under super low power mode.

After setting up the initial configurations for the ADC, call `adc_oneshot_new_unit()` with the prepared `adc_oneshot_unit_init_cfg_t`. This function will return an ADC unit handle, if the allocation is successful.

This function may fail due to various errors such as invalid argumemts, insufficient memory, etc. Specifically, when the to-be-allocated ADC instance is registered already, this function will return `ESP_ERR_NOT_FOUND` error. Number of available ADC(s) is recorded by `SOC_ADC_PERIPH_NUM`.

If a previously created ADC instance is no loger required, you should recycle the ADC instance by calling `adc_oneshot_del_unit()`, related hardware and software resources will be recycled as well.

Create an ADC Unit Handle under Normal Oneshot Mode

```
adc_oneshot_unit_handle_t adc1_handle;
adc_oneshot_unit_init_cfg_t init_config1 = {
    .unit_id = ADC_UNIT_1,
    .ulp_mode = ADC_ULP_MODE_DISABLE,
};
ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));
```

Recycle the ADC Unit

```
ESP_ERROR_CHECK(adc_oneshot_del_unit(adc1_handle));
```

Unit Configuration After an ADC instance is created, set up the `adc_oneshot_chan_cfg_t` to configure ADC IO to measure analog signal:

- `adc_oneshot_chan_cfg_t::atten`, ADC attenuation. Refer to the On-Chip Sensor chapter in TRM.
- `adc_oneshot_chan_cfg_t::channel`, the IO corresponding ADC channel number. See below note.
- `adc_oneshot_chan_cfg_t::bitwidth`, the bitwidth of the raw conversion result.

Note: For the IO corresponding ADC channel number. Check datasheet to know the ADC IOs. On the other hand, *adc_continuous_io_to_channel()* and *adc_continuous_channel_to_io()* can be used to know the ADC channels and ADC IOs.

To make these settings take effect, call *adc_oneshot_config_channel()* with above configuration structure. Especially, this *adc_oneshot_config_channel()* can be called multiple times to configure different ADC channels. Drvier will save these per channel configurations internally.

Configure Two ADC Channels

```
adc_oneshot_chan_cfg_t config = {
    .channel = EXAMPLE_ADC1_CHAN0,
    .bitwidth = ADC_BITWIDTH_DEFAULT,
    .atten = ADC_ATTEN_DB_12,
};
ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, &config));

config.channel = EXAMPLE_ADC1_CHAN1;
ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, &config));
```

Read Conversion Result After above configurations, the ADC is ready to measure the analog siganl(s) from the configured ADC channel(s). Call *adc_oneshot_read()* to get the conversion raw result of an ADC channel.

- *adc_oneshot_read()* is safer. ADC(s) are shared by some other drivers / peripherals, see *Hardware Limitations*. This function takes some mutexes, to avoid concurrent hardware usage. Therefore, this function should not be used in an ISR context. This function may fail when the ADC is in use by other drivers / peripherals, and return *ESP_ERR_TIMEOUT*. Under this condition, the ADC raw result is invalid.

These two functions will both fail due to invalid arguments.

The ADC conversion results read from these two functions are raw data. To calculate the voltage based on the ADC raw results, this formula can be used:

```
Vout = Dout * Vmax / Dmax        (1)
```

where:

Vout	Digital output result, standing for the voltage.
Dout	ADC raw digital reading result.
Vmax	Maximum measurable input analog voltage, this is related to the ADC attenuation, please refer to the On-Chip Sensor chapter in TRM.
Dmax	Maximum of the output ADC raw digital reading result, which is 2^bitwidth, where bitwidth is the :cpp:member::*adc_oneshot_chan_cfg_t:bitwidth* configured before.

To do further calbration to convert the ADC raw result to voltage in mV, please refer to calibration doc *Analog to Digital Converter (ADC) Calibration Driver*.

Read Raw Result

```
ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, EXAMPLE_ADC1_CHAN0, &adc_raw[0][0]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN0,
↪ adc_raw[0][0]);

ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, EXAMPLE_ADC1_CHAN1, &adc_raw[0][1]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN1,
↪ adc_raw[0][1]);
```

Hardware Limitations

- Random Number Generator uses ADC as a input source. When ADC `adc_oneshot_read()` works, the random number generated from RNG will be less random.
- A specific ADC unit can only work under one operating mode at any one time, either continuous mode or oneshot mode. `adc_oneshot_read()` has provided the protection.
- ADC2 is also used by the Wi-Fi. `adc_oneshot_read()` has provided the protection between Wi-Fi driver and ADC oneshot mode driver.
- ESP32 DevKitC: GPIO 0 cannot be used due to external auto program circuits.
- ESP-WROVER-KIT: GPIO 0, 2, 4 and 15 cannot be used due to external connections for different purposes.

Power Management When power management is enabled (i.e. *CONFIG_PM_ENABLE* is on), the system clock frequency may be adjusted when the system is in an idle state. However, the ADC oneshot mode driver works in a polling routine, the `adc_oneshot_read()` will poll the CPU until the function returns. During this period of time, the task in which ADC oneshot mode driver resides won't be blocked. Therefore the clock frequency is stable when reading.

IRAM Safe By default, all the ADC oneshot mode driver APIs are not supposed to be run when the Cache is disabled (Cache may be disabled due to many reasons, such as Flash writing/erasing, OTA, etc.). If these APIs executes when the Cache is disabled, you will probably see errors like Illegal Instruction or Load/Store Prohibited.

Thread Safety

- `adc_oneshot_new_unit()`
- `adc_oneshot_config_channel()`
- `adc_oneshot_read()`

Above functions are guaranteed to be thread safe. Therefore, you can call them from different RTOS tasks without protection by extra locks.

- `adc_oneshot_del_unit()` is not thread safe. Besides, concurrently calling this function may result in thread-safe APIs fail.

Kconfig Options

- *CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM* controls where to place the ADC fast read function (IRAM or Flash), see *IRAM Safe* for more details.

Application Examples

- ADC oneshot mode example: peripherals/adc/oneshot_read.

API Reference

Header File

- components/hal/include/hal/adc_types.h

Structures

struct **adc_digi_pattern_config_t**

ADC digital controller pattern configuration.

Public Members

uint8_t **atten**

Attenuation of this ADC channel.

uint8_t **channel**

ADC channel.

uint8_t **unit**

ADC unit.

uint8_t **bit_width**

ADC output bit width.

struct **adc_digi_output_data_t**

ADC digital controller (DMA mode) output data format. Used to analyze the acquired ADC (DMA) data.

Note: ESP32: Only `type1` is valid. ADC2 does not support DMA mode.

Note: ESP32-S2: Member `channel` can be used to judge the validity of the ADC data, because the role of the arbiter may get invalid ADC data.

Public Members

uint16_t **data**

ADC real output data info. Resolution: 12 bit.

ADC real output data info. Resolution: 11 bit.

uint16_t **channel**

ADC channel index info.

ADC channel index info. For ESP32-S2: If (channel < ADC_CHANNEL_MAX), The data is valid. If (channel > ADC_CHANNEL_MAX), The data is invalid.

struct *adc_digi_output_data_t*::[anonymous]::[anonymous] **type1**

ADC type1

uint16_t **unit**

ADC unit index info. 0: ADC1; 1: ADC2.

struct *adc_digi_output_data_t*::[anonymous]::[anonymous] **type2**

When the configured output format is 11bit.

uint16_t **val**

Raw data value

Enumerations

enum **adc_unit_t**

ADC unit.

Values:

enumerator **ADC_UNIT_1**

SAR ADC 1.

enumerator **ADC_UNIT_2**

SAR ADC 2.

enum **adc_channel_t**

ADC channels.

Values:

enumerator **ADC_CHANNEL_0**

ADC channel.

enumerator **ADC_CHANNEL_1**

ADC channel.

enumerator **ADC_CHANNEL_2**

ADC channel.

enumerator **ADC_CHANNEL_3**

ADC channel.

enumerator **ADC_CHANNEL_4**

ADC channel.

enumerator **ADC_CHANNEL_5**

ADC channel.

enumerator **ADC_CHANNEL_6**

ADC channel.

enumerator **ADC_CHANNEL_7**

ADC channel.

enumerator **ADC_CHANNEL_8**

ADC channel.

enumerator **ADC_CHANNEL_9**

ADC channel.

enum **adc_atten_t**

ADC attenuation parameter. Different parameters determine the range of the ADC.

Values:

enumerator **ADC_ATTEN_DB_0**

No input attenuation, ADC can measure up to approx.

enumerator **ADC_ATTEN_DB_2_5**

The input voltage of ADC will be attenuated extending the range of measurement by about 2.5 dB.

enumerator **ADC_ATTEN_DB_6**

The input voltage of ADC will be attenuated extending the range of measurement by about 6 dB.

enumerator **ADC_ATTEN_DB_12**

The input voltage of ADC will be attenuated extending the range of measurement by about 12 dB.

enumerator **ADC_ATTEN_DB_11**

This is deprecated, it behaves the same as `ADC_ATTEN_DB_12`

enum **adc_bitwidth_t**

Values:

enumerator **ADC_BITWIDTH_DEFAULT**

Default ADC output bits, max supported width will be selected.

enumerator **ADC_BITWIDTH_9**

ADC output width is 9Bit.

enumerator **ADC_BITWIDTH_10**

ADC output width is 10Bit.

enumerator **ADC_BITWIDTH_11**

ADC output width is 11Bit.

enumerator **ADC_BITWIDTH_12**

ADC output width is 12Bit.

enumerator **ADC_BITWIDTH_13**

ADC output width is 13Bit.

enum **adc_ulp_mode_t**

Values:

enumerator **ADC_ULP_MODE_DISABLE**

ADC ULP mode is disabled.

enumerator **ADC_ULP_MODE_FSM**

ADC is controlled by ULP FSM.

enumerator **ADC_ULP_MODE_RISCV**

ADC is controlled by ULP RISCV.

enum **adc_digi_convert_mode_t**

ADC digital controller (DMA mode) work mode.

Values:

enumerator **ADC_CONV_SINGLE_UNIT_1**

Only use ADC1 for conversion.

enumerator **ADC_CONV_SINGLE_UNIT_2**

Only use ADC2 for conversion.

enumerator **ADC_CONV_BOTH_UNIT**

Use Both ADC1 and ADC2 for conversion simultaneously.

enumerator **ADC_CONV_ALTER_UNIT**

Use both ADC1 and ADC2 for conversion by turn. e.g. ADC1 -> ADC2 -> ADC1 -> ADC2 ⋯..

enum **adc_digi_output_format_t**

ADC digital controller (DMA mode) output data format option.

Values:

enumerator **ADC_DIGI_OUTPUT_FORMAT_TYPE1**

See *adc_digi_output_data_t.type1*

enumerator **ADC_DIGI_OUTPUT_FORMAT_TYPE2**

See *adc_digi_output_data_t.type2*

Header File

- components/esp_adc/include/esp_adc/adc_oneshot.h

Functions

esp_err_t **adc_oneshot_new_unit**(const *adc_oneshot_unit_init_cfg_t* *init_config, *adc_oneshot_unit_handle_t* *ret_unit)

Create a handle to a specific ADC unit.

Note: This API is thread-safe. For more details, see ADC programming guide

Parameters

- **init_config** –**[in]** Driver initial configurations
- **ret_unit** –**[out]** ADC unit handle

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NO_MEM: No memory
- ESP_ERR_NOT_FOUND: The ADC peripheral to be claimed is already in use

esp_err_t **adc_oneshot_config_channel**(*adc_oneshot_unit_handle_t* handle, *adc_channel_t* channel, const *adc_oneshot_chan_cfg_t* *config)

Set ADC oneshot mode required configurations.

Note: This API is thread-safe. For more details, see ADC programming guide

Parameters
- **handle** –**[in]** ADC handle
- **channel** –**[in]** ADC channel to be configured
- **config** –**[in]** ADC configurations

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments

esp_err_t **adc_oneshot_read**(*adc_oneshot_unit_handle_t* handle, *adc_channel_t* chan, int *out_raw)

Get one ADC conversion raw result.

Note: This API is thread-safe. For more details, see ADC programming guide

Note: This API should NOT be called in an ISR context

Parameters
- **handle** –**[in]** ADC handle
- **chan** –**[in]** ADC channel
- **out_raw** –**[out]** ADC conversion raw result

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_TIMEOUT: Timeout, the ADC result is invalid

esp_err_t **adc_oneshot_del_unit**(*adc_oneshot_unit_handle_t* handle)

Delete the ADC unit handle.

Note: This API is thread-safe. For more details, see ADC programming guide

Parameters **handle** –**[in]** ADC handle

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NOT_FOUND: The ADC peripheral to be disclaimed isn’ t in use

esp_err_t **adc_oneshot_io_to_channel**(int io_num, *adc_unit_t* *unit_id, *adc_channel_t* *channel)

Get ADC channel from the given GPIO number.

Parameters
- **io_num** –**[in]** GPIO number
- **unit_id** –**[out]** ADC unit
- **channel** –**[out]** ADC channel

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NOT_FOUND: The IO is not a valid ADC pad

esp_err_t **adc_oneshot_channel_to_io**(*adc_unit_t* unit_id, *adc_channel_t* channel, int *io_num)

Get GPIO number from the given ADC channel.

Parameters

- **unit_id** –**[in]** ADC unit
- **channel** –**[in]** ADC channel
- **io_num** –**[out]** GPIO number
- **–** –ESP_OK: On success
  - ESP_ERR_INVALID_ARG: Invalid argument

Structures

struct **adc_oneshot_unit_init_cfg_t**

ADC oneshot driver initial configurations.

Public Members

adc_unit_t **unit_id**

ADC unit.

adc_ulp_mode_t **ulp_mode**

ADC controlled by ULP, see `adc_ulp_mode_t`

struct **adc_oneshot_chan_cfg_t**

ADC channel configurations.

Public Members

adc_atten_t **atten**

ADC attenuation.

adc_bitwidth_t **bitwidth**

ADC conversion result bits.

Type Definitions

typedef struct adc_oneshot_unit_ctx_t ***adc_oneshot_unit_handle_t**

Type of ADC unit handle for oneshot mode.

2.6.2 Analog to Digital Converter (ADC) Continuous Mode Driver

Introduction

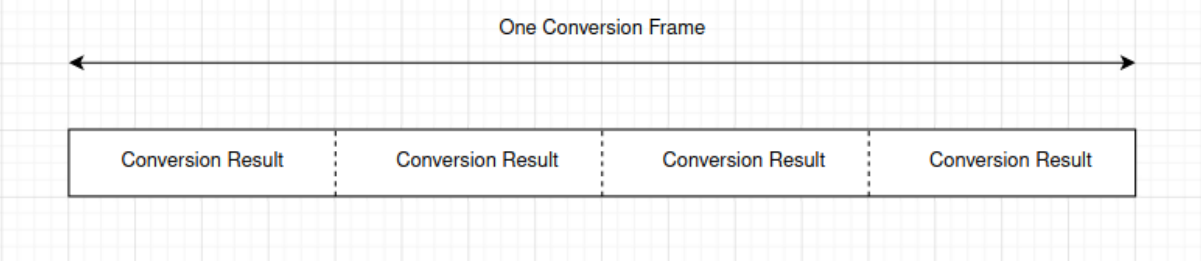
The Analog to Digital Converter is an on-chip sensor which is able to measure analog signals from specific analog IO pads.

The ADC on ESP32 can be used in scenario(s) like:

- Generate one-shot ADC conversion result
- Generate continuous ADC conversion results

This guide will introduce ADC continuous mode conversion.

Driver Concepts ADC continuous mode conversion is made up with multiple Conversion Frames. - Conversion Frame: One Conversion Frame contains multiple Conversion Results. Conversion Frame size is configured in `adc_continuous_new_handle()`, in bytes. - Conversion Result: One Conversion Result contains multiple bytes (see `SOC_ADC_DIGI_RESULT_BYTES`). Its structure is `adc_digi_output_data_t`, including ADC unit, ADC channel and raw data.



Functional Overview

The following sections of this document cover the typical steps to install the ADC continuous mode driver, and read ADC conversion results from group of ADC channels continuously:

- *Resource Allocation* - covers which parameters should be set up to initialize the ADC continuous mode driver and how to deinitialize it.
- *ADC Configurations* - describes how to configure the ADC(s) to make it work under continuous mode.
- *ADC Control* - describes ADC control functions.
- *Register Event Callbacks* - describes how to hook user specific code to an ADC continuous mode event callback function.
- *Read Conversion Result* - covers how to get ADC conversion result.
- *Hardware Limitations* - describes the ADC related hardware limitations.
- *Power Management* - covers power management related.
- *IRAM Safe* - covers the IRAM safe functions.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.

Resource Allocation The ADC continuous mode driver is implemented based on ESP32 SAR ADC module. Different ESP targets might have different number of independent ADCs.

To create an ADC continuous mode driver handle, set up the required configuration structure `adc_continuous_handle_cfg_t`:

- `adc_continuous_handle_cfg_t::max_store_buf_size` set the maximum size (in bytes) of the pool that the driver saves ADC conversion result into. If this pool is full, new conversion results will be lost.
- `adc_continuous_handle_cfg_t::conv_frame_size` set the size of the ADC conversion frame, in bytes.

After setting up above configurations for the ADC, call `adc_continuous_new_handle()` with the prepared `adc_continuous_handle_cfg_t`. This function may fail due to various errors such as invalid argumemts, insufficient memory, etc.

Especially, when this function returns `ESP_ERR_NOT_FOUND`, this means the I2S0 peripheral is in use. See *Hardware Limitations* for more information.

If the ADC continuous mode driver is no longer used, you should deinitialize the driver by calling `adc_continuous_deinit()`.

Initialize the ADC Continuous Mode Driver

```
adc_continuous_handle_cfg_t adc_config = {
    .max_store_buf_size = 1024,
    .conv_frame_size = 100,
```

(continues on next page)

(continued from previous page)

```
};
ESP_ERROR_CHECK(adc_continuous_new_handle(&adc_config));
```

Recycle the ADC Unit

```
ESP_ERROR_CHECK(adc_continuous_deinit());
```

ADC Configurations After the ADC continuous mode driver is initialized, set up the *adc_continuous_config_t* to configure ADC IOs to measure analog signal:

- *adc_continuous_config_t::pattern_num*, number of ADC channels that will be used.
- *adc_continuous_config_t::adc_pattern*, list of configs for each ADC channel that will be used, see below description.
- *adc_continuous_config_t::sample_freq_hz*, expected ADC sampling frequency in Hz.
- *adc_continuous_config_t::conv_mode*, continuous conversion mode.
- *adc_continuous_config_t::format*, conversion output format.

For *adc_digi_pattern_config_t*:

- *adc_digi_pattern_config_t::atten*, ADC attenuation. Refer to the On-Chip Sensor chapter in TRM.
- *adc_digi_pattern_config_t::channel*, the IO corresponding ADC channel number. See below note.
- *adc_digi_pattern_config_t::unit*, the ADC that the IO is subordinate to.
- *adc_digi_pattern_config_t::bit_width*, the bitwidth of the raw conversion result.

Note: For the IO corresponding ADC channel number. Check datasheet to acquire the ADC IOs. On the other hand, *adc_continuous_io_to_channel()* and *adc_continuous_channel_to_io()* can be used to acquire the ADC channels and ADC IOs.

To make these settings take effect, call *adc_continuous_config()* with the configuration structure above. This API may fail due to reasons like *ESP_ERR_INVALID_ARG*. When it returns *ESP_ERR_INVALID_STATE*, this means the ADC continuous mode driver is started, you shouldn't call this API at this moment.

See ADC continuous mode example peripherals/adc/continuous_read to see configuration codes.

ADC Control

Start and Stop Calling *adc_continuous_start()* will make the ADC start to measure analog signals from the configured ADC channels, and generate the conversion results. On the contrary, calling *adc_continuous_stop()* will stop the ADC conversion.

```
ESP_ERROR_CHECK(adc_continuous_stop());
```

Register Event Callbacks By calling *adc_continuous_register_event_callbacks()*, you can hook your own function to the driver ISR. Supported event callbacks are listed in *adc_continuous_evt_cbs_t* - *adc_continuous_evt_cbs_t::on_conv_done*, this is invoked when one conversion frame finishes. - *adc_continuous_evt_cbs_t::on_pool_ovf*, this is invoked when internal pool is full. Newer conversion results will be discarded.

As above callbacks are called in an ISR context, you should always ensure the callback function is suitable for an ISR context. Blocking logics should not appear in these callbacks. Callback function prototype is declared in *adc_continuous_callback_t*.

You can also register your own context when calling *adc_continuous_register_event_callbacks()*, by the parameter `user_data`. This user data will be passed to the callback functions directly.

This function may fail due to reasons like *ESP_ERR_INVALID_ARG*. Specially, when *CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE* is enabled, this error may indicate that the callback functions aren't in internal RAM. Check error log to know this. Besides, when it fails due to *ESP_ERR_INVALID_STATE*, this means the ADC continuous mode driver is started, you shouldn't add callback at this moment.

Conversion Done Event The driver will fill in the event data of a *adc_continuous_evt_cbs_t::on_conv_done* event. Event data contains a buffer pointer to a conversion frame buffer, together with the size. Refer to *adc_continuous_evt_data_t* to know the event data structure.

Note: It is worth noting that, the data buffer *adc_continuous_evt_data_t::conv_frame_buffer* is maintained by the driver itself. Therefore, never free this piece of memory.

Note: When the Kconfig option *CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE* is enabled, the registered callbacks and the functions called by the callbacks should be placed in IRAM. The involved variables should be placed in internal RAM as well.

Pool Overflow Event The ADC continuous mode driver has an internal pool to save the conversion results. When the pool is full, a pool overflow event will emerge. Under this condition, the driver won't fill in the event data. This usually happens the speed to read data from the pool (by calling *adc_continuous_read()*) is much slower than the ADC conversion speed.

Read Conversion Result After calling *adc_continuous_start()*, the ADC continuous conversion starts. Call *adc_continuous_read()* to get the conversion results of the ADC channels. You need to provide a buffer to get the raw results.

This function will try to read the expected length of conversion results each time.

- If the requested length isn't reached, the function will still move the data from the internal pool to the buffer you prepared. Therefore, check the *out_length* to know the actual size of conversion results.
- If there is no conversion result generated in the internal pool, the function will block for *timeout_ms* until the conversion results are generated. If there is still no generated results, the function will return *ESP_ERR_TIMEOUT*.
- If the generated results fill up the internal pool, new generated results will be lost. Next time when the *adc_continuous_read()* is called, this function will return *ESP_ERR_INVALID_STATE* indicating this situation.

This API aims to give you a chance to read all the ADC continuous conversion results.

The ADC conversion results read from above function are raw data. To calculate the voltage based on the ADC raw results, this formula can be used:

```
Vout = Dout * Vmax / Dmax        (1)
```

where:

Vout	Digital output result, standing for the voltage.
Dout	ADC raw digital reading result.
Vmax	Maximum measurable input analog voltage, this is related to the ADC attenuation, please refer to the On-Chip Sensor chapter in TRM.
Dmax	Maximum of the output ADC raw digital reading result, which is 2^bitwidth, where bitwidth is the :cpp:member::*adc_digi_pattern_config_t:bit_width* configured before.

To do further calbration to convert the ADC raw result to voltage in mV, please refer to calibration doc *Analog to Digital Converter (ADC) Calibration Driver*.

Hardware Limitations

- A specific ADC unit can only work under one operating mode at any one time, either continuous mode or oneshot mode. `adc_continuous_start()` has provided the protection.
- Random Number Generator uses ADC as an input source. When ADC continuous mode driver works, the random number generated from RNG will be less random.
- ADC2 is also used by the Wi-Fi. `adc_continuous_start()` has provided the protection between Wi-Fi driver and ADC continuous mode driver.
- ADC continuous mode driver uses I2S0 peripheral as hardware DMA fifo. Therefore, if I2S0 is in use already, the `adc_continuous_new_handle()` will return `ESP_ERR_NOT_FOUND`.
- ESP32 DevKitC: GPIO 0 cannot be used due to external auto program circuits.
- ESP-WROVER-KIT: GPIO 0, 2, 4 and 15 cannot be used due to external connections for different purposes.

Power Management When power management is enabled (i.e. *CONFIG_PM_ENABLE* is on), the APB clock frequency may be adjusted when the system is in an idle state, thus potentially changing the behavior of ADC continuous conversion.

However, the continuous mode driver can prevent this change by acquiring a power management lock of type `ESP_PM_APB_FREQ_MAX`. The lock is acquired after the continuous conversion is started by `adc_continuous_start()`. Similarly, the lock will be released after `adc_continuous_stop()`. Therefore, `adc_continuous_start()` and `adc_continuous_stop()` should appear in pairs, otherwise the power management will be out of action.

IRAM Safe All the ADC continuous mode driver APIs are not IRAM-safe. They are not supposed to be run when the Cache is disabled. By enabling the Kconfig option *CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE*, driver internal ISR handler is IRAM-safe, which means even when the Cache is disabled, the driver will still save the conversion results into its internal pool.

Thread Safety ADC continuous mode driver APIs are not guaranteed to be thread safe. However, the share hardware mutual exclusion is provided by the driver. See *Hardware Limitations* for more details.

Application Examples

- ADC continuous mode example: peripherals/adc/continuous_read.

API Reference

Header File

- components/esp_adc/include/esp_adc/adc_continuous.h

Functions

esp_err_t **adc_continuous_new_handle**(const *adc_continuous_handle_cfg_t* *hdl_config, *adc_continuous_handle_t* *ret_handle)

Initialize ADC continuous driver and get a handle to it.

Parameters

- **hdl_config** –**[in]** Pointer to ADC initilization config. Refer to `adc_continuous_handle_cfg_t`.
- **ret_handle** –**[out]** ADC continuous mode driver handle

Returns

- ESP_ERR_INVALID_ARG If the combination of arguments is invalid.
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_NO_MEM If out of memory
- ESP_OK On success

esp_err_t **adc_continuous_config**(*adc_continuous_handle_t* handle, const *adc_continuous_config_t* *config)

Set ADC continuous mode required configurations.

Parameters
- **handle** –**[in]** ADC continuous mode driver handle
- **config** –**[in]** Refer to `adc_digi_config_t`.

Returns
- ESP_ERR_INVALID_STATE: Driver state is invalid, you shouldn't call this API at this moment
- ESP_ERR_INVALID_ARG: If the combination of arguments is invalid.
- ESP_OK: On success

esp_err_t **adc_continuous_register_event_callbacks**(*adc_continuous_handle_t* handle, const *adc_continuous_evt_cbs_t* *cbs, void *user_data)

Register callbacks.

Note: User can deregister a previously registered callback by calling this function and setting the to-be-deregistered callback member int the `cbs` structure to NULL.

Note: When CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. Involved variables (including `user_data`) should be in internal RAM as well.

Note: You should only call this API when the ADC continuous mode driver isn't started. Check return value to know this.

Parameters
- **handle** –**[in]** ADC continuous mode driver handle
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be delivered to the callback functions directly

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Driver state is invalid, you shouldn't call this API at this moment

esp_err_t **adc_continuous_start**(*adc_continuous_handle_t* handle)

Start the ADC under continuous mode. After this, the hardware starts working.

Parameters **handle** –**[in]** ADC continuous mode driver handle

Returns
- ESP_ERR_INVALID_STATE Driver state is invalid.
- ESP_OK On success

esp_err_t **adc_continuous_read**(*adc_continuous_handle_t* handle, uint8_t *buf, uint32_t length_max, uint32_t *out_length, uint32_t timeout_ms)

Read bytes from ADC under continuous mode.

Parameters

- **handle** –**[in]** ADC continuous mode driver handle
- **buf** –**[out]** Conversion result buffer to read from ADC. Suggest convert to *adc_digi_output_data_t* for ADC Conversion Results. See @brief Driver Backgrounds to know this concept.
- **length_max** –**[in]** Expected length of the Conversion Results read from the ADC, in bytes.
- **out_length** –**[out]** Real length of the Conversion Results read from the ADC via this API, in bytes.
- **timeout_ms** –**[in]** Time to wait for data via this API, in millisecond.

Returns

- ESP_ERR_INVALID_STATE Driver state is invalid. Usually it means the ADC sampling rate is faster than the task processing rate.
- ESP_ERR_TIMEOUT Operation timed out
- ESP_OK On success

esp_err_t **adc_continuous_stop**(*adc_continuous_handle_t* handle)

Stop the ADC. After this, the hardware stops working.

Parameters **handle** –**[in]** ADC continuous mode driver handle

Returns

- ESP_ERR_INVALID_STATE Driver state is invalid.
- ESP_OK On success

esp_err_t **adc_continuous_deinit**(*adc_continuous_handle_t* handle)

Deinitialize the ADC continuous driver.

Parameters **handle** –**[in]** ADC continuous mode driver handle

Returns

- ESP_ERR_INVALID_STATE Driver state is invalid.
- ESP_OK On success

esp_err_t **adc_continuous_io_to_channel**(int io_num, *adc_unit_t* *unit_id, *adc_channel_t* *channel)

Get ADC channel from the given GPIO number.

Parameters

- **io_num** –**[in]** GPIO number
- **unit_id** –**[out]** ADC unit
- **channel** –**[out]** ADC channel

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NOT_FOUND: The IO is not a valid ADC pad

esp_err_t **adc_continuous_channel_to_io**(*adc_unit_t* unit_id, *adc_channel_t* channel, int *io_num)

Get GPIO number from the given ADC channel.

Parameters

- **unit_id** –**[in]** ADC unit
- **channel** –**[in]** ADC channel
- **io_num** –**[out]** GPIO number
- **–** –ESP_OK: On success
  - ESP_ERR_INVALID_ARG: Invalid argument

Structures

struct **adc_continuous_handle_cfg_t**

ADC continuous mode driver initial configurations.

Public Members

uint32_t **max_store_buf_size**

Max length of the conversion Results that driver can store, in bytes.

uint32_t **conv_frame_size**

Conversion frame size, in bytes. This should be in multiples of `SOC_ADC_DIGI_DATA_BYTES_PER_CONV`.

struct **adc_continuous_config_t**

ADC continuous mode driver configurations.

Public Members

uint32_t **pattern_num**

Number of ADC channels that will be used.

adc_digi_pattern_config_t ***adc_pattern**

List of configs for each ADC channel that will be used.

uint32_t **sample_freq_hz**

The expected ADC sampling frequency in Hz. Please refer to `soc/soc_caps.h` to know available sampling frequency range

adc_digi_convert_mode_t **conv_mode**

ADC DMA conversion mode, see `adc_digi_convert_mode_t`.

adc_digi_output_format_t **format**

ADC DMA conversion output format, see `adc_digi_output_format_t`.

struct **adc_continuous_evt_data_t**

Event data structure.

Note: The `conv_frame_buffer` is maintained by the driver itself, so never free this piece of memory.

Public Members

uint8_t ***conv_frame_buffer**

Pointer to conversion result buffer for one conversion frame.

uint32_t **size**

Conversion frame size.

struct **adc_continuous_evt_cbs_t**

Group of ADC continuous mode callbacks.

Note: These callbacks are all running in an ISR environment.

Note: When CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. Involved variables should be in internal RAM as well.

Public Members

adc_continuous_callback_t **on_conv_done**

Event callback, invoked when one conversion frame is done. See `@brief Driver Backgrounds` to konw `conversion frame` concept.

adc_continuous_callback_t **on_pool_ovf**

Event callback, invoked when the internal pool is full.

Macros

ADC_MAX_DELAY

Driver Backgrounds.

Type Definitions

typedef struct adc_continuous_ctx_t ***adc_continuous_handle_t**

Type of adc continuous mode driver handle.

typedef bool (***adc_continuous_callback_t**)(*adc_continuous_handle_t* handle, const *adc_continuous_evt_data_t* *edata, void *user_data)

Prototype of ADC continuous mode event callback.

Param handle **[in]** ADC continuous mode driver handle
Param edata **[in]** Pointer to ADC contunuous mode event data
Param user_data **[in]** User registered context, registered when in `adc_continuous_register_event_callbacks()`
Return Whether a high priority task is woken up by this function

2.6.3 Analog to Digital Converter (ADC) Calibration Driver

Introduction

Based on series of comparisons with the reference voltage, ESP32 ADC determines each bit of the output digital result. Per design the ESP32 ADC reference voltage is 1100 mV, however the true reference voltage can range from 1000 mV to 1200 mV among different chips. This guide will introduce an ADC calibration driver to minimize this difference.

Functional Overview

The following sections of this document cover the typical steps to install and use the ADC calibration driver:

- *Calibration Scheme Creation* - covers how to create a calibration scheme handle and delete the calibration scheme handle.
- *Calibration Configuration* - covers how to configure the calibration driver to calculate necessary characteristics used for calibration.
- *Result Conversion* - convers how to convert ADC raw result to calibrated result.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.
- *Minimize Noise* - describes a general way to minimize the noise.

- *Kconfig Options* - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

Calibration Scheme Creation The ADC calibration driver provides ADC calibration scheme(s). From calibration driver's point of view, an ADC calibration scheme is created to an ADC calibration handle *adc_cali_handle_t*.

adc_cali_check_scheme() can be used to know which calibration scheme is supported on the chip. For those users who are already aware of the supported scheme, this step can be skipped. Just call the corresponding function to create the scheme handle.

For those users who use their custom ADC calibration schemes, you could either modify this function *adc_cali_check_scheme()*, or just skip this step and call your custom creation function.

ADC Calibration Line Fitting Scheme ESP32 supports `ADC_CALI_SCHEME_VER_LINE_FITTING` scheme. To create this scheme, set up `adc_cali_line_fitting_config_t` first.

- `adc_cali_line_fitting_config_t::unit_id`, the ADC that your ADC raw results are from.
- `adc_cali_line_fitting_config_t::atten`, ADC attenuation that your ADC raw results use.
- `adc_cali_line_fitting_config_t::bitwidth`, the ADC raw result bitwidth.

There is also a configuration `adc_cali_line_fitting_config_t::default_vref`. Normally this can be simply set to 0. Line Fitting scheme doesn't rely on this value. However, if the Line Fitting scheme required eFuse bits are not burnt on your board, driver will rely on this value to do the calibration.

You can use `adc_cali_scheme_line_fitting_check_efuse()` to check the eFuse bits. Normally the Line Fitting scheme eFuse value will be `ADC_CALI_LINE_FITTING_EFUSE_VAL_EFUSE_TP` or `ADC_CALI_LINE_FITTING_EFUSE_VAL_EFUSE_VREF`. This means Line Fitting scheme will use calibration parameters burnt in the eFuse to do the calibration.

When the Line Fitting scheme eFuse value is `ADC_CALI_LINE_FITTING_EFUSE_VAL_DEFAULT_VREF`, you need to set the `esp_adc_cali_line_fitting_init::default_vref`. Default vref is an estimate of the ADC reference voltage provided by the users as a parameter during calibration.

After setting up the configuration structure, call `adc_cali_create_scheme_line_fitting()` to create a Line Fitting calibration scheme handle.

```
ESP_LOGI(TAG, "calibration scheme version is %s", "Line Fitting");
adc_cali_line_fitting_config_t cali_config = {
    .unit_id = unit,
    .atten = atten,
    .bitwidth = ADC_BITWIDTH_DEFAULT,
};
ESP_ERROR_CHECK(adc_cali_create_scheme_line_fitting(&cali_config, &handle));
```

When the ADC calibration is no longer used, please delete the calibration scheme handle by calling `adc_cali_delete_scheme_line_fitting()`.

Delete Line Fitting Scheme

```
ESP_LOGI(TAG, "delete %s calibration scheme", "Line Fitting");
ESP_ERROR_CHECK(adc_cali_delete_scheme_line_fitting(handle));
```

Note: For users who want to use their custom calibration schemes, you could provide a creation function to create your calibration scheme handle. Check the function table *adc_cali_scheme_t* in *components/esp_adc/interface/adc_cali_interface.h* to know the ESP ADC calibration interface.

Result Conversion After setting up the calibration characteristics, you can call *adc_cali_raw_to_voltage()* to convert the ADC raw result into calibrated result. The calibrated result is in the unit of mV. This function may fail due to invalid argument. Especaily, if this function returns *ESP_ERR_INVALID_STATE*, this means the calibration scheme isn't created. You need to create a calibration scheme handle, use *adc_cali_check_scheme()* to know the supported calibration scheme. On the other hand, you could also provide a custom calibration scheme and create the handle.

Get Voltage

```
ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adc_cali_handle, adc_raw[0][0], &
↪voltage[0][0]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV", ADC_UNIT_1 + 1, EXAMPLE_
↪ADC1_CHAN0, voltage[0][0]);
```

Thread Safety The factory function `esp_adc_cali_new_scheme()` is guaranteed to be thread safe by the driver. Therefore, you can call them from different RTOS tasks without protection by extra locks.

Other functions that take the *adc_cali_handle_t* as the first positional parameter are not thread safe, you should avoid calling them from multiple tasks.

Kconfig Options

- *CONFIG_ADC_CAL_EFUSE_TP_ENABLE*, disable this to decrease the code size, if you are aware of the calibration eFuse value `ADC_CALI_LINE_FITTING_EFUSE_VAL_EFUSE_TP` isn't this one.
- *CONFIG_ADC_CAL_EFUSE_VREF_ENABLE*, disable this to decrease the code size, if you are aware of the calibration eFuse value `ADC_CALI_LINE_FITTING_EFUSE_VAL_EFUSE_VREF` isn't this one.
- *CONFIG_ADC_CAL_LUT_ENABLE*, disable this to decrease the code size, if you don't calibrate the ADC raw results under `ADC_ATTEN_DB_12`.

Minimize Noise The ESP32 ADC can be sensitive to noise leading to large discrepancies in ADC readings. Depending on the usage scenario, you may need to connect a bypass capacitor (e.g. a 100 nF ceramic capacitor) to the ADC input pad in use, to minimize noise. Besides, multisampling may also be used to further mitigate the effects of noise.

API Reference

Header File

- components/esp_adc/include/esp_adc/adc_cali.h

Functions

esp_err_t **adc_cali_check_scheme**(*adc_cali_scheme_ver_t* *scheme_mask)

Check the supported ADC calibration scheme.

Parameters **scheme_mask** –**[out]** Supported ADC calibration scheme(s)

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NOT_SUPPORTED: No supported calibration scheme

esp_err_t **adc_cali_raw_to_voltage**(*adc_cali_handle_t* handle, int raw, int *voltage)

Convert ADC raw data to calibrated voltage.

Parameters

- **handle** –**[in]** ADC calibration handle
- **raw** –**[in]** ADC raw data
- **voltage** –**[out]** Calibrated ADC voltage (in mV)

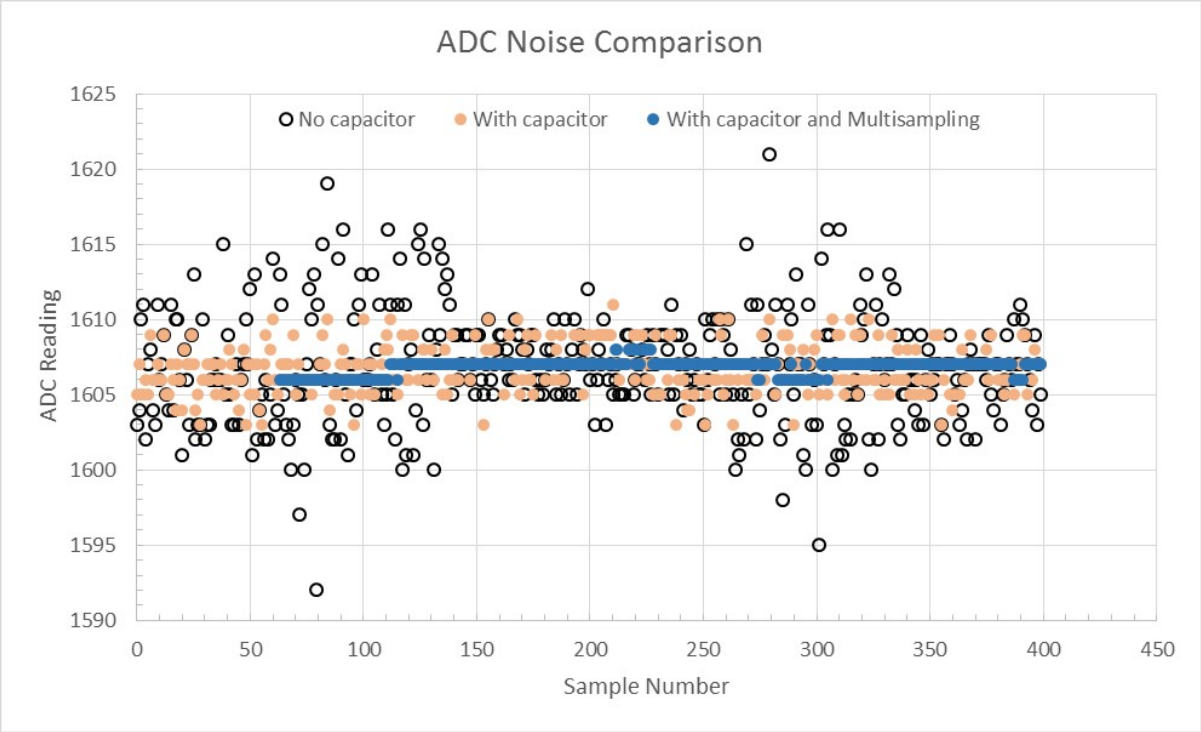


Fig. 6: Graph illustrating noise mitigation using capacitor and multisampling of 64 samples.

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Invalid state, scheme didn’ t registered

Type Definitions

typedef struct adc_cali_scheme_t ***adc_cali_handle_t**

ADC calibration handle.

Enumerations

enum **adc_cali_scheme_ver_t**

ADC calibration scheme.

Values:

enumerator **ADC_CALI_SCHEME_VER_LINE_FITTING**

Line fitting scheme.

enumerator **ADC_CALI_SCHEME_VER_CURVE_FITTING**

Curve fitting scheme.

Header File

- components/esp_adc/include/esp_adc/adc_cali_scheme.h

2.6.4 Clock Tree

This section lists definitions of the ESP32's supported root clocks and module clocks. These definitions are commonly used in the driver configuration, to help user select a proper source clock for the peripheral.

Root Clocks

Root clocks generate reliable clock signals. These clock signals then pass through various gates, muxes, dividers, or multipliers to become the clock sources for every functional module: the CPU core(s), WIFI, BT, the RTC, and the peripherals.

ESP32's root clocks are listed in *soc_root_clk_t*:

- Internal 8MHz RC Oscillator (RC_FAST)
  This RC oscillator generates a ~8.5MHz clock signal output as the RC_FAST_CLK.
  The ~8.5MHz signal output is also passed into a configurable divider, which by default divides the input clock frequency by 256, to generate a RC_FAST_D256_CLK.
  The exact frequency of RC_FAST_CLK can be computed in runtime through calibration on the RC_FAST_D256_CLK.
- External 2~40MHz Crystal (XTAL)
- Internal 150kHz RC Oscillator (RC_SLOW)
  This RC oscillator generates a ~150kHz clock signal output as the RC_SLOW_CLK.
  The exact frequency of this clock can be computed in runtime through calibration.
- External 32kHz Crystal - optional (XTAL32K)
  The clock source for this XTAL32K_CLK can be either a 32kHz crystal connecting to the 32K_XP and 32K_XN pins or a 32kHz clock signal generated by an external circuit.
  The external signal must be connected to the 32K_XN pin. Additionally, a 1nF capacitor must be placed between the 32K_XP pin and ground. In this case, the 32K_XP pin cannot be used as a GPIO pin.
  XTAL32K_CLK can also be calibrated to get its exact frequency.

Typically, the frequency of the signal generated from a RC oscillator circuit is less accurate and more sensitive to environment comparing to the signal generated from a crystal. ESP32 provides several clock source options for the RTC_SLOW_CLK, and users can make the choice based on the requirements for system time accuracy and power consumption (refer to *RTC Timer Clock Sources* for more details).

Module Clocks

ESP32's available module clocks are listed in *soc_module_clk_t*. Each module clock has a unique ID. You can get more information on each clock by checking the documented enum value.

API Reference

Header File

- components/soc/esp32/include/soc/clk_tree_defs.h

Macros

SOC_CLK_RC_FAST_FREQ_APPROX

Approximate RC_FAST_CLK frequency in Hz

SOC_CLK_RC_SLOW_FREQ_APPROX

Approximate RC_SLOW_CLK frequency in Hz

SOC_CLK_RC_FAST_D256_FREQ_APPROX

Approximate RC_FAST_D256_CLK frequency in Hz

SOC_CLK_XTAL32K_FREQ_APPROX

Approximate XTAL32K_CLK frequency in Hz

SOC_GPTIMER_CLKS

Array initializer for all supported clock sources of GPTimer.

The following code can be used to iterate all possible clocks:

```
soc_periph_gptimer_clk_src_t gptimer_clks[] = (soc_periph_gptimer_clk_src_
↪t)SOC_GPTIMER_CLKS;
for (size_t i = 0; i< sizeof(gptimer_clks) / sizeof(gptimer_clks[0]); i++) {
    soc_periph_gptimer_clk_src_t clk = gptimer_clks[i];
    // Test GPTimer with the clock `clk`
}
```

SOC_LCD_CLKS

Array initializer for all supported clock sources of LCD.

SOC_RMT_CLKS

Array initializer for all supported clock sources of RMT.

SOC_MCPWM_TIMER_CLKS

Array initializer for all supported clock sources of MCPWM Timer.

SOC_MCPWM_CAPTURE_CLKS

Array initializer for all supported clock sources of MCPWM Capture Timer.

SOC_MCPWM_CARRIER_CLKS

Array initializer for all supported clock sources of MCPWM Carrier.

SOC_I2S_CLKS

Array initializer for all supported clock sources of I2S.

SOC_I2C_CLKS

Array initializer for all supported clock sources of I2C.

SOC_SDM_CLKS

Array initializer for all supported clock sources of SDM.

Enumerations

enum **soc_root_clk_t**

Root clock.

Values:

enumerator **SOC_ROOT_CLK_INT_RC_FAST**

Internal 8MHz RC oscillator

enumerator **SOC_ROOT_CLK_INT_RC_SLOW**

Internal 150kHz RC oscillator

enumerator **SOC_ROOT_CLK_EXT_XTAL**

External 2~40MHz crystal

enumerator **SOC_ROOT_CLK_EXT_XTAL32K**

External 32kHz crystal/clock signal

enum **soc_cpu_clk_src_t**

CPU_CLK mux inputs, which are the supported clock sources for the CPU_CLK.

Note: Enum values are matched with the register field values on purpose

---

Values:

enumerator **SOC_CPU_CLK_SRC_XTAL**

Select XTAL_CLK as CPU_CLK source

enumerator **SOC_CPU_CLK_SRC_PLL**

Select PLL_CLK as CPU_CLK source (PLL_CLK is the output of 40MHz crystal oscillator frequency multiplier, can be 480MHz or 320MHz)

enumerator **SOC_CPU_CLK_SRC_RC_FAST**

Select RC_FAST_CLK as CPU_CLK source

enumerator **SOC_CPU_CLK_SRC_APLL**

Select APLL_CLK as CPU_CLK source

enumerator **SOC_CPU_CLK_SRC_INVALID**

Invalid CPU_CLK source

enum **soc_rtc_slow_clk_src_t**

RTC_SLOW_CLK mux inputs, which are the supported clock sources for the RTC_SLOW_CLK.

Note: Enum values are matched with the register field values on purpose

---

Values:

enumerator **SOC_RTC_SLOW_CLK_SRC_RC_SLOW**

Select RC_SLOW_CLK as RTC_SLOW_CLK source

enumerator **SOC_RTC_SLOW_CLK_SRC_XTAL32K**

Select XTAL32K_CLK as RTC_SLOW_CLK source

enumerator **SOC_RTC_SLOW_CLK_SRC_RC_FAST_D256**

Select RC_FAST_D256_CLK (referred as FOSC_DIV or 8m_d256/8md256 in TRM and reg. description) as RTC_SLOW_CLK source

enumerator **SOC_RTC_SLOW_CLK_SRC_INVALID**

Invalid RTC_SLOW_CLK source

enum **soc_rtc_fast_clk_src_t**

RTC_FAST_CLK mux inputs, which are the supported clock sources for the RTC_FAST_CLK.

Note: Enum values are matched with the register field values on purpose

Values:

enumerator **SOC_RTC_FAST_CLK_SRC_XTAL_D4**

Select XTAL_D4_CLK (may referred as XTAL_CLK_DIV_4) as RTC_FAST_CLK source

enumerator **SOC_RTC_FAST_CLK_SRC_XTAL_DIV**

Alias name for `SOC_RTC_FAST_CLK_SRC_XTAL_D4`

enumerator **SOC_RTC_FAST_CLK_SRC_RC_FAST**

Select RC_FAST_CLK as RTC_FAST_CLK source

enumerator **SOC_RTC_FAST_CLK_SRC_INVALID**

Invalid RTC_FAST_CLK source

enum **soc_module_clk_t**

Supported clock sources for modules (CPU, peripherals, RTC, etc.)

Note: enum starts from 1, to save 0 for special purpose

Values:

enumerator **SOC_MOD_CLK_CPU**

CPU_CLK can be sourced from XTAL, PLL, RC_FAST, or APLL by configuring soc_cpu_clk_src_t

enumerator **SOC_MOD_CLK_RTC_FAST**

RTC_FAST_CLK can be sourced from XTAL_D4 or RC_FAST by configuring soc_rtc_fast_clk_src_t

enumerator **SOC_MOD_CLK_RTC_SLOW**

RTC_SLOW_CLK can be sourced from RC_SLOW, XTAL32K, or RC_FAST_D256 by configuring soc_rtc_slow_clk_src_t

enumerator **SOC_MOD_CLK_APB**

APB_CLK is highly dependent on the CPU_CLK source

enumerator **SOC_MOD_CLK_PLL_D2**

PLL_D2_CLK is derived from PLL, it has a fixed divider of 2

enumerator **SOC_MOD_CLK_PLL_F160M**

PLL_F160M_CLK is derived from PLL, and has a fixed frequency of 160MHz

enumerator **SOC_MOD_CLK_XTAL32K**

XTAL32K_CLK comes from the external 32kHz crystal, passing a clock gating to the peripherals

enumerator **SOC_MOD_CLK_RC_FAST**

RC_FAST_CLK comes from the internal 8MHz rc oscillator, passing a clock gating to the peripherals

enumerator **SOC_MOD_CLK_RC_FAST_D256**

RC_FAST_D256_CLK comes from the internal 8MHz rc oscillator, divided by 256, and passing a clock gating to the peripherals

enumerator **SOC_MOD_CLK_XTAL**

XTAL_CLK comes from the external crystal (2~40MHz)

enumerator **SOC_MOD_CLK_REF_TICK**

REF_TICK is derived from APB, it has a fixed frequency of 1MHz even when APB frequency changes

enumerator **SOC_MOD_CLK_APLL**

APLL is sourced from PLL, and its frequency is configurable through APLL configuration registers

enum **soc_periph_gptimer_clk_src_t**

Type of GPTimer clock source.

Values:

enumerator **GPTIMER_CLK_SRC_APB**

Select APB as the source clock

enumerator **GPTIMER_CLK_SRC_DEFAULT**

Select APB as the default choice

enum **soc_periph_tg_clk_src_legacy_t**

Type of Timer Group clock source, reserved for the legacy timer group driver.

Values:

enumerator **TIMER_SRC_CLK_APB**

Timer group source clock is APB

enumerator **TIMER_SRC_CLK_DEFAULT**

Timer group source clock default choice is APB

enum **soc_periph_lcd_clk_src_t**

Type of LCD clock source.

Values:

enumerator **LCD_CLK_SRC_PLL160M**

Select PLL_160M as the source clock

enumerator **LCD_CLK_SRC_DEFAULT**

Select PLL_160M as the default choice

enum **soc_periph_rmt_clk_src_t**

Type of RMT clock source.

Values:

enumerator **RMT_CLK_SRC_APB**

Select APB as the source clock

enumerator **RMT_CLK_SRC_REF_TICK**

Select REF_TICK as the source clock

enumerator **RMT_CLK_SRC_DEFAULT**

Select APB as the default choice

enum **soc_periph_rmt_clk_src_legacy_t**

Type of RMT clock source, reserved for the legacy RMT driver.

Values:

enumerator **RMT_BASECLK_APB**

RMT source clock is APB CLK

enumerator **RMT_BASECLK_REF**

RMT source clock is REF_TICK

enumerator **RMT_BASECLK_DEFAULT**

RMT source clock default choice is APB

enum **soc_periph_uart_clk_src_legacy_t**

Type of UART clock source, reserved for the legacy UART driver.

Values:

enumerator **UART_SCLK_APB**

UART source clock is APB CLK

enumerator **UART_SCLK_REF_TICK**

UART source clock is REF_TICK

enumerator **UART_SCLK_DEFAULT**

UART source clock default choice is APB

enum **soc_periph_mcpwm_timer_clk_src_t**

Type of MCPWM timer clock source.

Values:

enumerator **MCPWM_TIMER_CLK_SRC_PLL160M**

Select PLL_F160M as the source clock

enumerator **MCPWM_TIMER_CLK_SRC_DEFAULT**

Select PLL_F160M as the default clock choice

enum **soc_periph_mcpwm_capture_clk_src_t**

Type of MCPWM capture clock source.

Values:

enumerator **MCPWM_CAPTURE_CLK_SRC_APB**

Select APB as the source clock

enumerator **MCPWM_CAPTURE_CLK_SRC_DEFAULT**

SElect APB as the default clock choice

enum **soc_periph_mcpwm_carrier_clk_src_t**

Type of MCPWM carrier clock source.

Values:

enumerator **MCPWM_CARRIER_CLK_SRC_PLL160M**

Select PLL_F160M as the source clock

enumerator **MCPWM_CARRIER_CLK_SRC_DEFAULT**

Select PLL_F160M as the default clock choice

enum **soc_periph_i2s_clk_src_t**

I2S clock source enum.

Values:

enumerator **I2S_CLK_SRC_DEFAULT**

Select PLL_D2 as the default source clock

enumerator **I2S_CLK_SRC_PLL_160M**

Select PLL_D2 as the source clock

enumerator **I2S_CLK_SRC_APLL**

Select APLL as the source clock

enum **soc_periph_i2c_clk_src_t**

Type of I2C clock source.

Values:

enumerator **I2C_CLK_SRC_APB**

enumerator **I2C_CLK_SRC_DEFAULT**

enum **soc_periph_sdm_clk_src_t**

Sigma Delta Modulator clock source.

Values:

enumerator **SDM_CLK_SRC_APB**

Select APB as the source clock

enumerator **SDM_CLK_SRC_DEFAULT**

Select APB as the default clock choice

2.6.5 Digital To Analog Converter (DAC)

Overview

ESP32 has two 8-bit DAC (digital to analog converter) channels, connected to GPIO25 (Channel 1) and GPIO26 (Channel 2).

The DAC driver allows these channels to be set to arbitrary voltages.

The DAC channels can also be driven with DMA-style written sample data by the digital controller, via the *I2S driver* when using the "built-in DAC mode".

For other analog output options, see the *Sigma-delta Modulation module* and the *LED Control module*. Both these modules produce high frequency PDM/PWM output, which can be hardware low-pass filtered in order to generate a lower frequency analog output.

Application Example

Setting DAC channel 1 (GPIO25) voltage to approx 0.78 of VDD_A voltage (VDD * 200 / 255). For VDD_A 3.3V, this is 2.59V.

```
#include <driver/dac.h>

...

dac_output_enable(DAC_CHANNEL_1);
dac_output_voltage(DAC_CHANNEL_1, 200);
```

API Reference

Header File

- components/driver/esp32/include/driver/dac.h

Functions

esp_err_t **dac_i2s_enable**(void)

Enable DAC output data from I2S.

Returns

- ESP_OK success

esp_err_t **dac_i2s_disable**(void)

Disable DAC output data from I2S.

Returns

- ESP_OK success

Header File

- components/driver/include/driver/dac_common.h

Functions

esp_err_t **dac_pad_get_io_num**(*dac_channel_t* channel, *gpio_num_t* *gpio_num)

Get the GPIO number of a specific DAC channel.

Parameters
- **channel** –Channel to get the gpio number
- **gpio_num** –output buffer to hold the gpio number

Returns
- ESP_OK if success

esp_err_t **dac_output_voltage**(*dac_channel_t* channel, uint8_t dac_value)

Set DAC output voltage. DAC output is 8-bit. Maximum (255) corresponds to VDD3P3_RTC.

Note: Need to configure DAC pad before calling this function. DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Parameters
- **channel** –DAC channel
- **dac_value** –DAC output value

Returns
- ESP_OK success

esp_err_t **dac_output_enable**(*dac_channel_t* channel)

DAC pad output enable.

Note: DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26 I2S left channel will be mapped to DAC channel 2 I2S right channel will be mapped to DAC channel 1

Parameters **channel** –DAC channel

esp_err_t **dac_output_disable**(*dac_channel_t* channel)

DAC pad output disable.

Note: DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Parameters **channel** –DAC channel

Returns
- ESP_OK success

esp_err_t **dac_cw_generator_enable**(void)

Enable cosine wave generator output.

Returns
- ESP_OK success

esp_err_t **dac_cw_generator_disable**(void)

Disable cosine wave generator output.

Returns
- ESP_OK success

esp_err_t **dac_cw_generator_config**(*dac_cw_config_t* *cw)

Config the cosine wave generator function in DAC module.

Parameters **cw** –Configuration.

Returns

- ESP_OK success
- ESP_ERR_INVALID_ARG The parameter is NULL.

GPIO Lookup Macros Some useful macros can be used to specified the GPIO number of a DAC channel, or vice versa. e.g.

1. `DAC_CHANNEL_1_GPIO_NUM` is the GPIO number of channel 1 (GPIO25);
2. `DAC_GPIO26_CHANNEL` is the channel number of GPIO 26 (channel 2).

Header File

- components/soc/esp32/include/soc/dac_channel.h

Macros

DAC_GPIO25_CHANNEL

DAC_CHANNEL_1_GPIO_NUM

DAC_GPIO26_CHANNEL

DAC_CHANNEL_2_GPIO_NUM

Header File

- components/hal/include/hal/dac_types.h

Structures

struct **dac_cw_config_t**

Config the cosine wave generator function in DAC module.

Public Members

dac_channel_t **en_ch**

Enable the cosine wave generator of DAC channel.

dac_cw_scale_t **scale**

Set the amplitude of the cosine wave generator output.

dac_cw_phase_t **phase**

Set the phase of the cosine wave generator output.

uint32_t **freq**

Set frequency of cosine wave generator output. Range: 130(130Hz) ~ 55000(100KHz).

int8_t **offset**

Set the voltage value of the DC component of the cosine wave generator output. Note: Unreasonable settings can cause waveform to be oversaturated. Range: -128 ~ 127.

Enumerations

enum **dac_channel_t**

Values:

enumerator **DAC_CHANNEL_1**

DAC channel 1 is GPIO25(ESP32) / GPIO17(ESP32S2)

enumerator **DAC_CHANNEL_2**

DAC channel 2 is GPIO26(ESP32) / GPIO18(ESP32S2)

enumerator **DAC_CHANNEL_MAX**

enum **dac_cw_scale_t**

The multiple of the amplitude of the cosine wave generator. The max amplitude is VDD3P3_RTC.

Values:

enumerator **DAC_CW_SCALE_1**

1/1. Default.

enumerator **DAC_CW_SCALE_2**

1/2.

enumerator **DAC_CW_SCALE_4**

1/4.

enumerator **DAC_CW_SCALE_8**

1/8.

enum **dac_cw_phase_t**

Set the phase of the cosine wave generator output.

Values:

enumerator **DAC_CW_PHASE_0**

Phase shift +0°

enumerator **DAC_CW_PHASE_180**

Phase shift +180°

2.6.6 GPIO & RTC GPIO

Overview

The ESP32 chip features 34 physical GPIO pins (GPIO0 ~ GPIO19, GPIO21 ~ GPIO23, GPIO25 ~ GPIO27, and GPIO32 ~ GPIO39). Each pin can be used as a general-purpose I/O, or be connected to an internal peripheral signal. Through IO MUX, RTC IO MUX and the GPIO matrix, peripheral input signals can be from any IO pins, and peripheral output signals can be routed to any IO pins. Together these modules provide highly configurable I/O. For more details, see *ESP32 Technical Reference Manual* > *IO MUX and GPIO Matrix (GPIO, IO_MUX)* [PDF].

The table below provides more information on pin usage, and please note the comments in the table for GPIOs with restrictions.

GPIO	Analog Function	RTC GPIO	Comments
GPIO0	ADC2_CH1	RTC_GPIO11	Strapping pin
GPIO1			TXD
GPIO2	ADC2_CH2	RTC_GPIO12	Strapping pin
GPIO3			RXD
GPIO4	ADC2_CH0	RTC_GPIO10	
GPIO5			Strapping pin
GPIO6			SPI0/1
GPIO7			SPI0/1
GPIO8			SPI0/1
GPIO9			SPI0/1
GPIO10			SPI0/1
GPIO11			SPI0/1
GPIO12	ADC2_CH5	RTC_GPIO15	Strapping pin; JTAG
GPIO13	ADC2_CH4	RTC_GPIO14	JTAG
GPIO14	ADC2_CH6	RTC_GPIO16	JTAG
GPIO15	ADC2_CH3	RTC_GPIO13	Strapping pin; JTAG
GPIO16			SPI0/1
GPIO17			SPI0/1
GPIO18			
GPIO19			
GPIO21			
GPIO22			
GPIO23			
GPIO25	ADC2_CH8	RTC_GPIO6	
GPIO26	ADC2_CH9	RTC_GPIO7	
GPIO27	ADC2_CH7	RTC_GPIO17	
GPIO32	ADC1_CH4	RTC_GPIO9	
GPIO33	ADC1_CH5	RTC_GPIO8	
GPIO34	ADC1_CH6	RTC_GPIO4	GPI
GPIO35	ADC1_CH7	RTC_GPIO5	GPI
GPIO36	ADC1_CH0	RTC_GPIO0	GPI
GPIO37	ADC1_CH1	RTC_GPIO1	GPI
GPIO38	ADC1_CH2	RTC_GPIO2	GPI
GPIO39	ADC1_CH3	RTC_GPIO3	GPI

Note:

- Strapping pin: GPIO0, GPIO2, GPIO5, GPIO12 (MTDI), and GPIO15 (MTDO) are strapping pins. For more infomation, please refer to ESP32 datasheet.
- SPI0/1: GPIO6-11 and GPIO16-17 are usually connected to the SPI flash and PSRAM integrated on the module and therefore should not be used for other purposes.
- JTAG: GPIO12-15 are usually used for inline debug.
- GPI: GPIO34-39 can only be set as input mode and do not have software-enabled pullup or pulldown functions.
- TXD & RXD are usually used for flashing and debugging.
- ADC2: ADC2 pins cannot be used when Wi-Fi is used. So, if you are having trouble getting the value from an ADC2 GPIO while using Wi-Fi, you may consider using an ADC1 GPIO instead, which should solve your problem. For more details, please refer to ADC limitations.

There is also separate "RTC GPIO" support, which functions when GPIOs are routed to the "RTC" low-power and analog subsystem. These pin functions can be used when:

- In Deep-sleep mode
- The *Ultra Low Power co-processor* is running
- Analog functions such as ADC/DAC/etc are in use.

Application Example

GPIO output and input interrupt example: peripherals/gpio/generic_gpio.

API Reference - Normal GPIO

Header File

- components/driver/include/driver/gpio.h

Functions

esp_err_t **gpio_config**(const *gpio_config_t* *pGPIOConfig)

GPIO common configuration.

```
Configure GPIO's Mode,pull-up,PullDown,IntrType
```

Parameters **pGPIOConfig** –Pointer to GPIO configure struct

Returns

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_reset_pin**(*gpio_num_t* gpio_num)

Reset an gpio to default state (select gpio function, enable pullup and disable input and output).

Note: This function also configures the IOMUX for this pin to the GPIO function, and disconnects any other peripheral output configured via GPIO Matrix.

Parameters **gpio_num** –GPIO number.

Returns Always return ESP_OK.

esp_err_t **gpio_set_intr_type**(*gpio_num_t* gpio_num, *gpio_int_type_t* intr_type)

GPIO set interrupt trigger type.

Parameters

- **gpio_num** –GPIO number. If you want to set the trigger type of e.g. of GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **intr_type** –Interrupt type, select from gpio_int_type_t

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_intr_enable**(*gpio_num_t* gpio_num)

Enable GPIO module interrupt signal.

Note: ESP32: Please do not use the interrupt of GPIO36 and GPIO39 when using ADC or Wi-Fi and Bluetooth with sleep mode enabled. Please refer to the comments of `adc1_get_raw`. Please refer to Section 3.11 of ESP32 ECO and Workarounds for Bugs for the description of this issue. As a workaround, call adc_power_acquire() in the app. This will result in higher power consumption (by ~1mA), but will remove the glitches on GPIO36 and GPIO39.

Parameters **gpio_num** –GPIO number. If you want to enable an interrupt on e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_intr_disable** (*gpio_num_t* gpio_num)

Disable GPIO module interrupt signal.

Note: This function is allowed to be executed when Cache is disabled within ISR context, by enabling CONFIG_GPIO_CTRL_FUNC_IN_IRAM

Parameters **gpio_num** –GPIO number. If you want to disable the interrupt of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

Returns

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_set_level** (*gpio_num_t* gpio_num, uint32_t level)

GPIO set output level.

Note: This function is allowed to be executed when Cache is disabled within ISR context, by enabling CONFIG_GPIO_CTRL_FUNC_IN_IRAM

Parameters

- **gpio_num** –GPIO number. If you want to set the output level of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **level** –Output level. 0: low ; 1: high

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO number error

int **gpio_get_level** (*gpio_num_t* gpio_num)

GPIO get input level.

Warning: If the pad is not configured for input (or input and output) the returned value is always 0.

Parameters **gpio_num** –GPIO number. If you want to get the logic level of e.g. pin GPIO16, gpio_num should be GPIO_NUM_16 (16);

Returns

- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

esp_err_t **gpio_set_direction** (*gpio_num_t* gpio_num, *gpio_mode_t* mode)

GPIO set direction.

Configure GPIO direction,such as output_only,input_only,output_and_input

Parameters

- **gpio_num** –Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **mode** –GPIO direction

Returns

- ESP_OK Success

- ESP_ERR_INVALID_ARG GPIO error

esp_err_t **gpio_set_pull_mode**(*gpio_num_t* gpio_num, *gpio_pull_mode_t* pull)

Configure GPIO pull-up/pull-down resistors.

Note: ESP32: Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

Parameters
- **gpio_num** –GPIO number. If you want to set pull up or down mode for e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **pull** –GPIO pull up/down mode.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG : Parameter error

esp_err_t **gpio_wakeup_enable**(*gpio_num_t* gpio_num, *gpio_int_type_t* intr_type)

Enable GPIO wake-up function.

Parameters
- **gpio_num** –GPIO number.
- **intr_type** –GPIO wake-up type. Only GPIO_INTR_LOW_LEVEL or GPIO_INTR_HIGH_LEVEL can be used.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_wakeup_disable**(*gpio_num_t* gpio_num)

Disable GPIO wake-up function.

Parameters **gpio_num** –GPIO number

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_isr_register**(void (*fn)(void*), void *arg, int intr_alloc_flags, *gpio_isr_handle_t* *handle)

Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

This ISR function is called whenever any GPIO interrupt occurs. See the alternative gpio_install_isr_service() and gpio_isr_handler_add() API in order to have the driver support per-GPIO ISRs.

To disable or remove the ISR, pass the returned handle to the *interrupt allocation functions*.

Parameters
- **fn** –Interrupt handler function.
- **arg** –Parameter for handler function
- **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- **handle** –Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Returns
- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags

esp_err_t **gpio_pullup_en**(*gpio_num_t* gpio_num)

Enable pull-up on GPIO.

Parameters **gpio_num** –GPIO number

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_pullup_dis**(*gpio_num_t* gpio_num)

Disable pull-up on GPIO.

Parameters **gpio_num** –GPIO number
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_pulldown_en**(*gpio_num_t* gpio_num)

Enable pull-down on GPIO.

Parameters **gpio_num** –GPIO number
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_pulldown_dis**(*gpio_num_t* gpio_num)

Disable pull-down on GPIO.

Parameters **gpio_num** –GPIO number
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_install_isr_service**(int intr_alloc_flags)

Install the GPIO driver's ETS_GPIO_INTR_SOURCE ISR handler service, which allows per-pin GPIO interrupt handlers.

This function is incompatible with gpio_isr_register() - if that function is used, a single global ISR is registered for all GPIO interrupts. If this function is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the gpio_isr_handler_add() function.

Parameters **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
Returns
- ESP_OK Success
- ESP_ERR_NO_MEM No memory to install this service
- ESP_ERR_INVALID_STATE ISR service already installed.
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_INVALID_ARG GPIO error

void **gpio_uninstall_isr_service**(void)

Uninstall the driver's GPIO ISR service, freeing related resources.

esp_err_t **gpio_isr_handler_add**(*gpio_num_t* gpio_num, *gpio_isr_t* isr_handler, void *args)

Add ISR handler for the corresponding GPIO pin.

Call this function after using gpio_install_isr_service() to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with IRAM_ATTR, unless you pass the ESP_INTR_FLAG_IRAM flag when allocating the ISR in gpio_install_isr_service().

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as "ISR stack size" in menuconfig). This limit is smaller compared to a global GPIO interrupt handler due to the additional level of indirection.

Parameters
- **gpio_num** –GPIO number
- **isr_handler** –ISR handler function for the corresponding GPIO number.
- **args** –parameter for ISR handler.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_isr_handler_remove**(*gpio_num_t* gpio_num)

Remove ISR handler for the corresponding GPIO pin.

Parameters **gpio_num** –GPIO number

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_set_drive_capability**(*gpio_num_t* gpio_num, *gpio_drive_cap_t* strength)

Set GPIO pad drive capability.

Parameters

- **gpio_num** –GPIO number, only support output GPIOs
- **strength** –Drive capability of the pad

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_get_drive_capability**(*gpio_num_t* gpio_num, *gpio_drive_cap_t* *strength)

Get GPIO pad drive capability.

Parameters

- **gpio_num** –GPIO number, only support output GPIOs
- **strength** –Pointer to accept drive capability of the pad

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **gpio_hold_en**(*gpio_num_t* gpio_num)

Enable gpio pad hold function.

When the pin is set to hold, the state is latched at that moment and will not change no matter how the internal signals change or how the IO MUX/GPIO configuration is modified (including input enable, output enable, output value, function, and drive strength values). It can be used to retain the pin state through a core reset and system reset triggered by watchdog time-out or Deep-sleep events.

The gpio pad hold function works in both input and output modes, but must be output-capable gpios. If pad hold enabled: in output mode: the output level of the pad will be force locked and can not be changed. in input mode: input read value can still reflect the changes of the input signal.

The state of the digital gpio cannot be held during Deep-sleep, and it will resume to hold at its default pin state when the chip wakes up from Deep-sleep. If the digital gpio also needs to be held during Deep-sleep, `gpio_deep_sleep_hold_en` should also be called.

Power down or call `gpio_hold_dis` will disable this function.

Parameters **gpio_num** –GPIO number, only support output-capable GPIOs

Returns

- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

esp_err_t **gpio_hold_dis**(*gpio_num_t* gpio_num)

Disable gpio pad hold function.

When the chip is woken up from Deep-sleep, the gpio will be set to the default mode, so, the gpio will output the default level if this function is called. If you don't want the level changes, the gpio should be configured to a known state before this function is called. e.g. If you hold gpio18 high during Deep-sleep, after the chip is woken up and `gpio_hold_dis` is called, gpio18 will output low level(because gpio18 is input mode by

default). If you don't want this behavior, you should configure gpio18 as output mode and set it to hight level before calling `gpio_hold_dis`.

Parameters **`gpio_num`** –GPIO number, only support output-capable GPIOs
Returns
- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

void **`gpio_deep_sleep_hold_en`**(void)

Enable all digital gpio pads hold function during Deep-sleep.

Enabling this feature makes all digital gpio pads be at the holding state during Deep-sleep. The state of each pad holds is its active configuration (not pad's sleep configuration!).

Note that this pad hold feature only works when the chip is in Deep-sleep mode. When the chip is in active mode, the digital gpio state can be changed freely even you have called this function.

After this API is being called, the digital gpio Deep-sleep hold feature will work during every sleep process. You should call `gpio_deep_sleep_hold_dis` to disable this feature.

void **`gpio_deep_sleep_hold_dis`**(void)

Disable all digital gpio pads hold function during Deep-sleep.

void **`gpio_iomux_in`**(uint32_t gpio_num, uint32_t signal_idx)

Set pad input to a peripheral signal through the IOMUX.

Parameters
- **`gpio_num`** –GPIO number of the pad.
- **`signal_idx`** –Peripheral signal id to input. One of the `*_IN_IDX` signals in `soc/gpio_sig_map.h`.

void **`gpio_iomux_out`**(uint8_t gpio_num, int func, bool oen_inv)

Set peripheral output to an GPIO pad through the IOMUX.

Parameters
- **`gpio_num`** –gpio_num GPIO number of the pad.
- **`func`** –The function number of the peripheral pin to output pin. One of the `FUNC_X_*` of specified pin (X) in `soc/io_mux_reg.h`.
- **`oen_inv`** –True if the output enable needs to be inverted, otherwise False.

esp_err_t **`gpio_sleep_sel_en`**(*gpio_num_t* gpio_num)

Enable SLP_SEL to change GPIO status automantically in lightsleep.

Parameters **`gpio_num`** –GPIO number of the pad.
Returns
- ESP_OK Success

esp_err_t **`gpio_sleep_sel_dis`**(*gpio_num_t* gpio_num)

Disable SLP_SEL to change GPIO status automantically in lightsleep.

Parameters **`gpio_num`** –GPIO number of the pad.
Returns
- ESP_OK Success

esp_err_t **`gpio_sleep_set_direction`**(*gpio_num_t* gpio_num, *gpio_mode_t* mode)

GPIO set direction at sleep.

Configure GPIO direction,such as output_only,input_only,output_and_input

Parameters
- **`gpio_num`** –Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **`mode`** –GPIO direction

Returns
- ESP_OK Success

- ESP_ERR_INVALID_ARG GPIO error

esp_err_t **gpio_sleep_set_pull_mode**(*gpio_num_t* gpio_num, *gpio_pull_mode_t* pull)

Configure GPIO pull-up/pull-down resistors at sleep.

Note: ESP32: Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

Parameters

- **gpio_num** –GPIO number. If you want to set pull up or down mode for e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- **pull** –GPIO pull up/down mode.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG : Parameter error

Structures

struct **gpio_config_t**

Configuration parameters of GPIO pad for gpio_config function.

Public Members

uint64_t **pin_bit_mask**

GPIO pin: set with bit mask, each bit maps to a GPIO

gpio_mode_t **mode**

GPIO mode: set input/output mode

gpio_pullup_t **pull_up_en**

GPIO pull-up

gpio_pulldown_t **pull_down_en**

GPIO pull-down

gpio_int_type_t **intr_type**

GPIO interrupt type

Macros

GPIO_PIN_COUNT

GPIO_IS_VALID_GPIO(gpio_num)

Check whether it is a valid GPIO number.

GPIO_IS_VALID_OUTPUT_GPIO(gpio_num)

Check whether it can be a valid GPIO number of output mode.

GPIO_IS_VALID_DIGITAL_IO_PAD(gpio_num)

Check whether it can be a valid digital I/O pad.

Type Definitions

typedef *intr_handle_t* **gpio_isr_handle_t**

typedef void (***gpio_isr_t**)(void *arg)

GPIO interrupt handler.

Param arg User registered data

Header File

- components/hal/include/hal/gpio_types.h

Macros

GPIO_PIN_REG_0

GPIO_PIN_REG_1

GPIO_PIN_REG_2

GPIO_PIN_REG_3

GPIO_PIN_REG_4

GPIO_PIN_REG_5

GPIO_PIN_REG_6

GPIO_PIN_REG_7

GPIO_PIN_REG_8

GPIO_PIN_REG_9

GPIO_PIN_REG_10

GPIO_PIN_REG_11

GPIO_PIN_REG_12

GPIO_PIN_REG_13

GPIO_PIN_REG_14

GPIO_PIN_REG_15

GPIO_PIN_REG_16

GPIO_PIN_REG_17

GPIO_PIN_REG_18

GPIO_PIN_REG_19

GPIO_PIN_REG_20

GPIO_PIN_REG_21

GPIO_PIN_REG_22

GPIO_PIN_REG_23

GPIO_PIN_REG_24

GPIO_PIN_REG_25

GPIO_PIN_REG_26

GPIO_PIN_REG_27

GPIO_PIN_REG_28

GPIO_PIN_REG_29

GPIO_PIN_REG_30

GPIO_PIN_REG_31

GPIO_PIN_REG_32

GPIO_PIN_REG_33

GPIO_PIN_REG_34

GPIO_PIN_REG_35

GPIO_PIN_REG_36

GPIO_PIN_REG_37

GPIO_PIN_REG_38

GPIO_PIN_REG_39

GPIO_PIN_REG_40

GPIO_PIN_REG_41

GPIO_PIN_REG_42

GPIO_PIN_REG_43

GPIO_PIN_REG_44

GPIO_PIN_REG_45

GPIO_PIN_REG_46

GPIO_PIN_REG_47

GPIO_PIN_REG_48

Enumerations

enum **gpio_port_t**

Values:

enumerator **GPIO_PORT_0**

enumerator **GPIO_PORT_MAX**

enum **gpio_num_t**

Values:

enumerator **GPIO_NUM_NC**

Use to signal not connected to S/W

enumerator **GPIO_NUM_0**

GPIO0, input and output

enumerator **GPIO_NUM_1**

GPIO1, input and output

enumerator **GPIO_NUM_2**

GPIO2, input and output

enumerator **GPIO_NUM_3**

GPIO3, input and output

enumerator **GPIO_NUM_4**

GPIO4, input and output

enumerator **GPIO_NUM_5**
GPIO5, input and output

enumerator **GPIO_NUM_6**
GPIO6, input and output

enumerator **GPIO_NUM_7**
GPIO7, input and output

enumerator **GPIO_NUM_8**
GPIO8, input and output

enumerator **GPIO_NUM_9**
GPIO9, input and output

enumerator **GPIO_NUM_10**
GPIO10, input and output

enumerator **GPIO_NUM_11**
GPIO11, input and output

enumerator **GPIO_NUM_12**
GPIO12, input and output

enumerator **GPIO_NUM_13**
GPIO13, input and output

enumerator **GPIO_NUM_14**
GPIO14, input and output

enumerator **GPIO_NUM_15**
GPIO15, input and output

enumerator **GPIO_NUM_16**
GPIO16, input and output

enumerator **GPIO_NUM_17**
GPIO17, input and output

enumerator **GPIO_NUM_18**
GPIO18, input and output

enumerator **GPIO_NUM_19**
GPIO19, input and output

enumerator **GPIO_NUM_20**
GPIO20, input and output

enumerator **GPIO_NUM_21**
GPIO21, input and output

enumerator **GPIO_NUM_22**
GPIO22, input and output

enumerator **GPIO_NUM_23**
GPIO23, input and output

enumerator **GPIO_NUM_25**
GPIO25, input and output

enumerator **GPIO_NUM_26**
GPIO26, input and output

enumerator **GPIO_NUM_27**
GPIO27, input and output

enumerator **GPIO_NUM_28**
GPIO28, input and output

enumerator **GPIO_NUM_29**
GPIO29, input and output

enumerator **GPIO_NUM_30**
GPIO30, input and output

enumerator **GPIO_NUM_31**
GPIO31, input and output

enumerator **GPIO_NUM_32**
GPIO32, input and output

enumerator **GPIO_NUM_33**
GPIO33, input and output

enumerator **GPIO_NUM_34**
GPIO34, input mode only

enumerator **GPIO_NUM_35**
GPIO35, input mode only

enumerator **GPIO_NUM_36**
GPIO36, input mode only

enumerator **GPIO_NUM_37**
GPIO37, input mode only

enumerator **GPIO_NUM_38**

GPIO38, input mode only

enumerator **GPIO_NUM_39**

GPIO39, input mode only

enumerator **GPIO_NUM_MAX**

enum **gpio_int_type_t**

Values:

enumerator **GPIO_INTR_DISABLE**

Disable GPIO interrupt

enumerator **GPIO_INTR_POSEDGE**

GPIO interrupt type : rising edge

enumerator **GPIO_INTR_NEGEDGE**

GPIO interrupt type : falling edge

enumerator **GPIO_INTR_ANYEDGE**

GPIO interrupt type : both rising and falling edge

enumerator **GPIO_INTR_LOW_LEVEL**

GPIO interrupt type : input low level trigger

enumerator **GPIO_INTR_HIGH_LEVEL**

GPIO interrupt type : input high level trigger

enumerator **GPIO_INTR_MAX**

enum **gpio_mode_t**

Values:

enumerator **GPIO_MODE_DISABLE**

GPIO mode : disable input and output

enumerator **GPIO_MODE_INPUT**

GPIO mode : input only

enumerator **GPIO_MODE_OUTPUT**

GPIO mode : output only mode

enumerator **GPIO_MODE_OUTPUT_OD**

GPIO mode : output only with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT_OD**

GPIO mode : output and input with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT**

GPIO mode : output and input mode

enum **gpio_pullup_t**

Values:

enumerator **GPIO_PULLUP_DISABLE**

Disable GPIO pull-up resistor

enumerator **GPIO_PULLUP_ENABLE**

Enable GPIO pull-up resistor

enum **gpio_pulldown_t**

Values:

enumerator **GPIO_PULLDOWN_DISABLE**

Disable GPIO pull-down resistor

enumerator **GPIO_PULLDOWN_ENABLE**

Enable GPIO pull-down resistor

enum **gpio_pull_mode_t**

Values:

enumerator **GPIO_PULLUP_ONLY**

Pad pull up

enumerator **GPIO_PULLDOWN_ONLY**

Pad pull down

enumerator **GPIO_PULLUP_PULLDOWN**

Pad pull up + pull down

enumerator **GPIO_FLOATING**

Pad floating

enum **gpio_drive_cap_t**

Values:

enumerator **GPIO_DRIVE_CAP_0**

Pad drive capability: weak

enumerator **GPIO_DRIVE_CAP_1**

Pad drive capability: stronger

enumerator **GPIO_DRIVE_CAP_2**

Pad drive capability: medium

enumerator **GPIO_DRIVE_CAP_DEFAULT**

Pad drive capability: medium

enumerator **GPIO_DRIVE_CAP_3**

Pad drive capability: strongest

enumerator **GPIO_DRIVE_CAP_MAX**

API Reference - RTC GPIO

Header File

- components/driver/include/driver/rtc_io.h

Functions

bool **rtc_gpio_is_valid_gpio**(*gpio_num_t* gpio_num)

Determine if the specified GPIO is a valid RTC GPIO.

Parameters **gpio_num** –GPIO number
Returns true if GPIO is valid for RTC GPIO use. false otherwise.

int **rtc_io_number_get**(*gpio_num_t* gpio_num)

Get RTC IO index number by gpio number.

Parameters **gpio_num** –GPIO number
Returns >=0: Index of rtcio. -1 : The gpio is not rtcio.

esp_err_t **rtc_gpio_init**(*gpio_num_t* gpio_num)

Init a GPIO as RTC GPIO.

This function must be called when initializing a pad for an analog function.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_deinit**(*gpio_num_t* gpio_num)

Init a GPIO as digital GPIO.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

uint32_t **rtc_gpio_get_level**(*gpio_num_t* gpio_num)

Get the RTC IO input level.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- 1 High level
- 0 Low level
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_set_level**(*gpio_num_t* gpio_num, uint32_t level)

Set the RTC IO output level.

Parameters
- **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
- **level** –output level

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_set_direction**(*gpio_num_t* gpio_num, *rtc_gpio_mode_t* mode)

RTC GPIO set direction.

Configure RTC GPIO direction, such as output only, input only, output and input.

Parameters
- **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
- **mode** –GPIO direction

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_set_direction_in_sleep**(*gpio_num_t* gpio_num, *rtc_gpio_mode_t* mode)

RTC GPIO set direction in deep sleep mode or disable sleep status (default). In some application scenarios, IO needs to have another states during deep sleep.

NOTE: ESP32 support INPUT_ONLY mode. ESP32S2 support INPUT_ONLY, OUTPUT_ONLY, INPUT_OUTPUT mode.

Parameters
- **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
- **mode** –GPIO direction

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_pullup_en**(*gpio_num_t* gpio_num)

RTC GPIO pullup enable.

This function only works for RTC IOs. In general, call gpio_pullup_en, which will work both for normal GPIOs and RTC IOs.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_pulldown_en**(*gpio_num_t* gpio_num)

RTC GPIO pulldown enable.

This function only works for RTC IOs. In general, call gpio_pulldown_en, which will work both for normal GPIOs and RTC IOs.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_pullup_dis**(*gpio_num_t* gpio_num)

RTC GPIO pullup disable.

This function only works for RTC IOs. In general, call gpio_pullup_dis, which will work both for normal GPIOs and RTC IOs.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_pulldown_dis** (*gpio_num_t* gpio_num)

RTC GPIO pulldown disable.

This function only works for RTC IOs. In general, call gpio_pulldown_dis, which will work both for normal GPIOs and RTC IOs.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_set_drive_capability** (*gpio_num_t* gpio_num, *gpio_drive_cap_t* strength)

Set RTC GPIO pad drive capability.

Parameters
- **gpio_num** –GPIO number, only support output GPIOs
- **strength** –Drive capability of the pad

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **rtc_gpio_get_drive_capability** (*gpio_num_t* gpio_num, *gpio_drive_cap_t* *strength)

Get RTC GPIO pad drive capability.

Parameters
- **gpio_num** –GPIO number, only support output GPIOs
- **strength** –Pointer to accept drive capability of the pad

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **rtc_gpio_hold_en** (*gpio_num_t* gpio_num)

Enable hold function on an RTC IO pad.

Enabling HOLD function will cause the pad to latch current values of input enable, output enable, output value, function, drive strength values. This function is useful when going into light or deep sleep mode to prevent the pin configuration from changing.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_hold_dis** (*gpio_num_t* gpio_num)

Disable hold function on an RTC IO pad.

Disabling hold function will allow the pad receive the values of input enable, output enable, output value, function, drive strength from RTC_IO peripheral.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12)
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

esp_err_t **rtc_gpio_isolate** (*gpio_num_t* gpio_num)

Helper function to disconnect internal circuits from an RTC IO This function disables input, output, pullup, pulldown, and enables hold feature for an RTC IO. Use this function if an RTC IO needs to be disconnected from internal circuits in deep sleep, to minimize leakage current.

In particular, for ESP32-WROVER module, call rtc_gpio_isolate(GPIO_NUM_12) before entering deep sleep, to reduce deep sleep current.

Parameters **gpio_num** –GPIO number (e.g. GPIO_NUM_12).
Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if GPIO is not an RTC IO

esp_err_t **rtc_gpio_force_hold_en_all**(void)

Enable force hold signal for all RTC IOs.

Each RTC pad has a "force hold" input signal from the RTC controller. If this signal is set, pad latches current values of input enable, function, output enable, and other signals which come from the RTC mux. Force hold signal is enabled before going into deep sleep for pins which are used for EXT1 wakeup.

esp_err_t **rtc_gpio_force_hold_dis_all**(void)

Disable force hold signal for all RTC IOs.

esp_err_t **rtc_gpio_wakeup_enable**(*gpio_num_t* gpio_num, *gpio_int_type_t* intr_type)

Enable wakeup from sleep mode using specific GPIO.

Parameters
- **gpio_num** –GPIO number
- **intr_type** –Wakeup on high level (GPIO_INTR_HIGH_LEVEL) or low level (GPIO_INTR_LOW_LEVEL)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if gpio_num is not an RTC IO, or intr_type is not one of GPIO_INTR_HIGH_LEVEL, GPIO_INTR_LOW_LEVEL.

esp_err_t **rtc_gpio_wakeup_disable**(*gpio_num_t* gpio_num)

Disable wakeup from sleep mode using specific GPIO.

Parameters **gpio_num** –GPIO number

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if gpio_num is not an RTC IO

Macros

RTC_GPIO_IS_VALID_GPIO(gpio_num)

Header File

- components/hal/include/hal/rtc_io_types.h

Enumerations

enum **rtc_gpio_mode_t**

RTCIO output/input mode type.

Values:

enumerator **RTC_GPIO_MODE_INPUT_ONLY**

Pad input

enumerator **RTC_GPIO_MODE_OUTPUT_ONLY**

Pad output

enumerator **RTC_GPIO_MODE_INPUT_OUTPUT**

Pad input + output

enumerator **RTC_GPIO_MODE_DISABLED**

Pad (output + input) disable

enumerator **RTC_GPIO_MODE_OUTPUT_OD**

Pad open-drain output

enumerator **RTC_GPIO_MODE_INPUT_OUTPUT_OD**

Pad input + open-drain output

2.6.7 General Purpose Timer (GPTimer)

Introduction

GPTimer (General Purpose Timer) is the driver of ESP32 Timer Group peripheral. The hardware timer features high resolution and flexible alarm action. The behavior when the internal counter of a timer reaches a specific target value is called a timer alarm. When a timer alarms, a user registered per-timer callback would be called.

Typically, a general purpose timer can be used in scenarios like:

- Free running as a wall clock, fetching a high-resolution timestamp at any time and any places
- Generate period alarms, trigger events periodically
- Generate one-shot alarm, respond in target time

Functional Overview

The following sections of this document cover the typical steps to install and operate a timer:

- *Resource Allocation* - covers which parameters should be set up to get a timer handle and how to recycle the resources when GPTimer finishes working.
- *Set and Get Count Value* - covers how to force the timer counting from a start point and how to get the count value at anytime.
- *Set up Alarm Action* - covers the parameters that should be set up to enable the alarm event.
- *Register Event Callbacks* - covers how to hook user specific code to the alarm event callback function.
- *Enable and Disable Timer* - covers how to enable and disable the timer.
- *Start and Stop Timer* - shows some typical use cases that start the timer with different alarm behavior.
- *Power Management* - describes how different source clock selections can affect power consumption.
- *IRAM Safe* - describes tips on how to make the timer interrupt and IO control functions work better along with a disabled cache.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.
- *Kconfig Options* - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

Resource Allocation Different ESP chips might have different numbers of independent timer groups, and within each group, there could also be several independent timers.[1]

A GPTimer instance is represented by `gptimer_handle_t`. The driver behind will manage all available hardware resources in a pool, so that you do not need to care about which timer and which group it belongs to.

To install a timer instance, there is a configuration structure that needs to be given in advance: `gptimer_config_t`:

- `gptimer_config_t::clk_src` selects the source clock for the timer. The available clocks are listed in `gptimer_clock_source_t`, you can only pick one of them. For the effect on power consumption of different clock source, please refer to Section *Power Management*.

[1] Different ESP chip series might have different numbers of GPTimer instances. For more details, please refer to *ESP32 Technical Reference Manual* > Chapter *Timer Group (TIMG)* [PDF]. The driver will not forbid you from applying for more timers, but it will return error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g. `gptimer_new_timer()`).

- *gptimer_config_t::direction* sets the counting direction of the timer, supported directions are listed in *gptimer_count_direction_t*, you can only pick one of them.
- *gptimer_config_t::resolution_hz* sets the resolution of the internal counter. Each count step is equivalent to **1 / resolution_hz** seconds.
- gptimer_config::intr_priority sets the priority of the timer interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.
- Optional *gptimer_config_t::intr_shared* sets whether or not mark the timer interrupt source as a shared one. For the pros/cons of a shared interrupt, you can refer to *Interrupt Handling*.

With all the above configurations set in the structure, the structure can be passed to *gptimer_new_timer()* which will instantiate the timer instance and return a handle of the timer.

The function can fail due to various errors such as insufficient memory, invalid arguments, etc. Specifically, when there are no more free timers (i.e. all hardware resources have been used up), then *ESP_ERR_NOT_FOUND* will be returned. The total number of available timers is represented by the *SOC_TIMER_GROUP_TOTAL_TIMERS* and its value will depend on the ESP chip.

If a previously created GPTimer instance is no longer required, you should recycle the timer by calling *gptimer_del_timer()*. This will allow the underlying HW timer to be used for other purposes. Before deleting a GPTimer handle, please disable it by *gptimer_disable()* in advance or make sure it has not enabled yet by *gptimer_enable()*.

Creating a GPTimer Handle with Resolution of 1 MHz

```
gptimer_handle_t gptimer = NULL;
gptimer_config_t timer_config = {
    .clk_src = GPTIMER_CLK_SRC_DEFAULT,
    .direction = GPTIMER_COUNT_UP,
    .resolution_hz = 1 * 1000 * 1000, // 1MHz, 1 tick = 1us
};
ESP_ERROR_CHECK(gptimer_new_timer(&timer_config, &gptimer));
```

Set and Get Count Value When the GPTimer is created, the internal counter will be reset to zero by default. The counter value can be updated asynchronously by *gptimer_set_raw_count()*. The maximum count value is dependent on the bit width of the hardware timer, which is also reflected by the SOC macro *SOC_TIMER_GROUP_COUNTER_BIT_WIDTH*. When updating the raw count of an active timer, the timer will immediately start counting from the new value.

Count value can be retrieved by *gptimer_get_raw_count()*, at any time.

Set up Alarm Action For most of the use cases of GPTimer, you should set up the alarm action before starting the timer, except for the simple wall-clock scenario, where a free running timer is enough. To set up the alarm action, you should configure several members of *gptimer_alarm_config_t* based on how you make use of the alarm event:

- *gptimer_alarm_config_t::alarm_count* sets the target count value that will trigger the alarm event. You should also take the counting direction into consideration when setting the alarm value. Specially, *gptimer_alarm_config_t::alarm_count* and *gptimer_alarm_config_t::reload_count* cannot be set to the same value when *gptimer_alarm_config_t::auto_reload_on_alarm* is true, as keeping reload with a target alarm count is meaningless.
- *gptimer_alarm_config_t::reload_count* sets the count value to be reloaded when the alarm event happens. This configuration only takes effect when *gptimer_alarm_config_t::auto_reload_on_alarm* is set to true.
- *gptimer_alarm_config_t::auto_reload_on_alarm* flag sets whether to enable the auto-reload feature. If enabled, the hardware timer will reload the value of *gptimer_alarm_config_t::reload_count* into counter immediately when an alarm event happens.

To make the alarm configurations take effect, you should call *gptimer_set_alarm_action()*. Especially, if *gptimer_alarm_config_t* is set to NULL, the alarm function will be disabled.

Note: If an alarm value is set and the timer has already exceeded this value, the alarm will be triggered immediately.

Register Event Callbacks After the timer starts up, it can generate a specific event (e.g. the "Alarm Event") dynamically. If you have some functions that should be called when the event happens, please hook your function to the interrupt service routine by calling *gptimer_register_event_callbacks()*. All supported event callbacks are listed in *gptimer_event_callbacks_t*:

- *gptimer_event_callbacks_t::on_alarm* sets a callback function for alarm events. As this function is called within the ISR context, you must ensure that the function does not attempt to block (e.g., by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The function prototype is declared in *gptimer_alarm_cb_t*.

You can save your own context to *gptimer_register_event_callbacks()* as well, via the parameter `user_data`. The user data will be directly passed to the callback function.

This function will lazy install the interrupt service for the timer but not enable it. So please call this function before *gptimer_enable()*, otherwise the *ESP_ERR_INVALID_STATE* error will be returned. See Section *Enable and Disable Timer* for more information.

Enable and Disable Timer Before doing IO control to the timer, you needs to enable the timer first, by calling *gptimer_enable()*. This function will:

- Switch the timer driver state from **init** to **enable**.
- Enable the interrupt service if it has been lazy installed by *gptimer_register_event_callbacks()*.
- Acquire a proper power management lock if a specific clock source (e.g. APB clock) is selected. See Section *Power Management* for more information.

Calling *gptimer_disable()* will do the opposite, that is, put the timer driver back to the **init** state, disable the interrupts service and release the power management lock.

Start and Stop Timer The basic IO operation of a timer is to start and stop. Calling *gptimer_start()* can make the internal counter work, while calling *gptimer_stop()* can make the counter stop working. The following illustrates how to start a timer with or without an alarm event. Calling *gptimer_start()* will transit the driver state from **enable** to **run**, and vice versa. You need to make sure the start and stop functions are used in pairs, otherwise, the functions may return *ESP_ERR_INVALID_STATE*. Most of the time, this error means that the timer is already stopped or in the "start protection" state (i.e. *gptimer_start()* is called but not finished).

Start Timer as a Wall Clock

```
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer));
// Retrieve the timestamp at any time
uint64_t count;
ESP_ERROR_CHECK(gptimer_get_raw_count(gptimer, &count));
```

Trigger Period Events

```
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm_
↪event_data_t *edata, void *user_ctx)
{
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_ctx;
```

(continues on next page)

(continued from previous page)

```
    // Retrieve the count value from event data
    example_queue_element_t ele = {
        .event_count = edata->count_value
    };
    // Optional: send the event data to other task by OS queue
    // Do not introduce complex logics in callbacks
    // Suggest dealing with event data in the main loop, instead of in this␣
↪callback
    xQueueSendFromISR(queue, &ele, &high_task_awoken);
    // return whether we need to yield at the end of ISR
    return high_task_awoken == pdTRUE;
}

gptimer_alarm_config_t alarm_config = {
    .reload_count = 0, // counter will reload with 0 on alarm event
    .alarm_count = 1000000, // period = 1s @resolution 1MHz
    .flags.auto_reload_on_alarm = true, // enable auto-reload
};
ESP_ERROR_CHECK(gptimer_set_alarm_action(gptimer, &alarm_config));

gptimer_event_callbacks_t cbs = {
    .on_alarm = example_timer_on_alarm_cb, // register user callback
};
ESP_ERROR_CHECK(gptimer_register_event_callbacks(gptimer, &cbs, queue));
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer));
```

Trigger One-Shot Event

```
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm_
↪event_data_t *edata, void *user_ctx)
{
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_ctx;
    // Stop timer the sooner the better
    gptimer_stop(timer);
    // Retrieve the count value from event data
    example_queue_element_t ele = {
        .event_count = edata->count_value
    };
    // Optional: send the event data to other task by OS queue
    xQueueSendFromISR(queue, &ele, &high_task_awoken);
    // return whether we need to yield at the end of ISR
    return high_task_awoken == pdTRUE;
}

gptimer_alarm_config_t alarm_config = {
    .alarm_count = 1 * 1000 * 1000, // alarm target = 1s @resolution 1MHz
};
ESP_ERROR_CHECK(gptimer_set_alarm_action(gptimer, &alarm_config));

gptimer_event_callbacks_t cbs = {
    .on_alarm = example_timer_on_alarm_cb, // register user callback
};
ESP_ERROR_CHECK(gptimer_register_event_callbacks(gptimer, &cbs, queue));
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer));
```

Dynamic Alarm Update Alarm value can be updated dynamically inside the ISR handler callback, by changing *gptimer_alarm_event_data_t::alarm_value*. Then the alarm value will be updated after the callback function returns.

```
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm_
↪event_data_t *edata, void *user_ctx)
{
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_data;
    // Retrieve the count value from event data
    example_queue_element_t ele = {
        .event_count = edata->count_value
    };
    // Optional: send the event data to other task by OS queue
    xQueueSendFromISR(queue, &ele, &high_task_awoken);
    // reconfigure alarm value
    gptimer_alarm_config_t alarm_config = {
        .alarm_count = edata->alarm_value + 1000000, // alarm in next 1s
    };
    gptimer_set_alarm_action(timer, &alarm_config);
    // return whether we need to yield at the end of ISR
    return high_task_awoken == pdTRUE;
}

gptimer_alarm_config_t alarm_config = {
    .alarm_count = 1000000, // initial alarm target = 1s @resolution 1MHz
};
ESP_ERROR_CHECK(gptimer_set_alarm_action(gptimer, &alarm_config));

gptimer_event_callbacks_t cbs = {
    .on_alarm = example_timer_on_alarm_cb, // register user callback
};
ESP_ERROR_CHECK(gptimer_register_event_callbacks(gptimer, &cbs, queue));
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer, &alarm_config));
```

Power Management There're some power management strategies, which might turn off or change the frequency of GPTimer's source clock to save power consumption. For example, during DFS, APB clock will be scaled down. If light-sleep is also enabled, PLL and XTAL clocks will be powered off. Both of them can result in an inaccurate time keeping.

The driver can prevent the above situation from happening by creating different power management lock according to different clock source. The driver will increase the reference count of that power management lock in the *gptimer_enable()* and decrease it in the *gptimer_disable()*. So we can ensure the clock source is stable between *gptimer_enable()* and *gptimer_disable()*.

IRAM Safe By default, the GPTimer interrupt will be deferred when the cache is disabled because of writing or erasing the flash. Thus the alarm interrupt will not get executed in time, which is not expected in a real-time application.

There is a Kconfig option *CONFIG_GPTIMER_ISR_IRAM_SAFE* that will:

- Enable the interrupt being serviced even when the cache is disabled
- Place all functions that used by the ISR into IRAM[2]
- Place driver object into DRAM (in case it is mapped to PSRAM by accident)

[2] *gptimer_event_callbacks_t::on_alarm* callback and the functions invoked by the callback should also be placed in IRAM, please take care of them by yourself.

This will allow the interrupt to run while the cache is disabled, but will come at the cost of increased IRAM consumption.

There is another Kconfig option *CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM* that can put commonly used IO control functions into IRAM as well. So, these functions can also be executable when the cache is disabled. These IO control functions are as follows:

- *gptimer_start()*
- *gptimer_stop()*
- *gptimer_get_raw_count()*
- *gptimer_set_raw_count()*
- *gptimer_set_alarm_action()*

Thread Safety All the APIs provided by the driver are guaranteed to be thread safe, which means you can call them from different RTOS tasks without protection by extra locks. The following functions are allowed to run under ISR context.

- *gptimer_start()*
- *gptimer_stop()*
- *gptimer_get_raw_count()*
- *gptimer_set_raw_count()*
- *gptimer_set_alarm_action()*

Kconfig Options

- *CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM* controls where to place the GPTimer control functions (IRAM or flash), see Section *IRAM Safe* for more information.
- *CONFIG_GPTIMER_ISR_IRAM_SAFE* controls whether the default ISR handler can work when the cache is disabled, see Section *IRAM Safe* for more information.
- *CONFIG_GPTIMER_ENABLE_DEBUG_LOG* is used to enabled the debug log output. Enable this option will increase the firmware binary size.

Application Examples

- Typical use cases of GPTimer are listed in the example peripherals/timer_group/gptimer.

API Reference

Header File

- components/driver/include/driver/gptimer.h

Functions

esp_err_t **gptimer_new_timer**(const *gptimer_config_t* *config, *gptimer_handle_t* *ret_timer)

Create a new General Purpose Timer, and return the handle.

Note: The newly created timer is put in the "init" state.

Parameters

- **config** –**[in]** GPTimer configuration
- **ret_timer** –**[out]** Returned timer handle

Returns

- ESP_OK: Create GPTimer successfully
- ESP_ERR_INVALID_ARG: Create GPTimer failed because of invalid argument
- ESP_ERR_NO_MEM: Create GPTimer failed because out of memory

- ESP_ERR_NOT_FOUND: Create GPTimer failed because all hardware timers are used up and no more free one
- ESP_FAIL: Create GPTimer failed because of other error

esp_err_t **gptimer_del_timer**(*gptimer_handle_t* timer)

Delete the GPTimer handle.

Note: A timer must be in the "init" state before it can be deleted.

Parameters **timer** –**[in]** Timer handle created by `gptimer_new_timer()`

Returns

- ESP_OK: Delete GPTimer successfully
- ESP_ERR_INVALID_ARG: Delete GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete GPTimer failed because the timer is not in init state
- ESP_FAIL: Delete GPTimer failed because of other error

esp_err_t **gptimer_set_raw_count**(*gptimer_handle_t* timer, uint64_t value)

Set GPTimer raw count value.

Note: When updating the raw count of an active timer, the timer will immediately start counting from the new value.

Note: This function is allowed to run within ISR context

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

Parameters

- **timer** –**[in]** Timer handle created by `gptimer_new_timer()`
- **value** –**[in]** Count value to be set

Returns

- ESP_OK: Set GPTimer raw count value successfully
- ESP_ERR_INVALID_ARG: Set GPTimer raw count value failed because of invalid argument
- ESP_FAIL: Set GPTimer raw count value failed because of other error

esp_err_t **gptimer_get_raw_count**(*gptimer_handle_t* timer, uint64_t *value)

Get GPTimer raw count value.

Note: With the raw count value and the resolution set in the *gptimer_config_t*, you can convert the count value into seconds.

Note: This function is allowed to run within ISR context

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

Parameters

- **timer** –**[in]** Timer handle created by `gptimer_new_timer()`
- **value** –**[out]** Returned GPTimer count value

Returns

- ESP_OK: Get GPTimer raw count value successfully
- ESP_ERR_INVALID_ARG: Get GPTimer raw count value failed because of invalid argument
- ESP_FAIL: Get GPTimer raw count value failed because of other error

esp_err_t **gptimer_register_event_callbacks**(*gptimer_handle_t* timer, const *gptimer_event_callbacks_t* *cbs, void *user_data)

Set callbacks for GPTimer.

Note: User registered callbacks are expected to be runnable within ISR context

Note: The first call to this function needs to be before the call to `gptimer_enable`

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **timer** –**[in]** Timer handle created by `gptimer_new_timer()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because the timer is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **gptimer_set_alarm_action**(*gptimer_handle_t* timer, const *gptimer_alarm_config_t* *config)

Set alarm event actions for GPTimer.

Note: This function is allowed to run within ISR context, so that user can set new alarm action immediately in the ISR callback.

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

Parameters

- **timer** –**[in]** Timer handle created by `gptimer_new_timer()`
- **config** –**[in]** Alarm configuration, especially, set config to NULL means disabling the alarm function

Returns

- ESP_OK: Set alarm action for GPTimer successfully
- ESP_ERR_INVALID_ARG: Set alarm action for GPTimer failed because of invalid argument
- ESP_FAIL: Set alarm action for GPTimer failed because of other error

esp_err_t **gptimer_enable** (*gptimer_handle_t* timer)

Enable GPTimer.

Note: This function will transit the timer state from "init" to "enable".

Note: This function will enable the interrupt service, if it's lazy installed in `gptimer_register_event_callbacks`.

Note: This function will acquire a PM lock, if a specific source clock (e.g. APB) is selected in the *gptimer_config_t*, while `CONFIG_PM_ENABLE` is enabled.

Note: Enable a timer doesn't mean to start it. See also `gptimer_start()` for how to make the timer start counting.

Parameters **timer** –**[in]** Timer handle created by `gptimer_new_timer()`

Returns

- ESP_OK: Enable GPTimer successfully
- ESP_ERR_INVALID_ARG: Enable GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable GPTimer failed because the timer is already enabled
- ESP_FAIL: Enable GPTimer failed because of other error

esp_err_t **gptimer_disable** (*gptimer_handle_t* timer)

Disable GPTimer.

Note: This function will transit the timer state from "enable" to "init".

Note: This function will disable the interrupt service if it's installed.

Note: This function will release the PM lock if it's acquired in the `gptimer_enable`.

Note: Disable a timer doesn't mean to stop it. See also `gptimer_stop` for how to make the timer stop counting.

Parameters **timer** –**[in]** Timer handle created by `gptimer_new_timer()`

Returns

- ESP_OK: Disable GPTimer successfully
- ESP_ERR_INVALID_ARG: Disable GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable GPTimer failed because the timer is not enabled yet
- ESP_FAIL: Disable GPTimer failed because of other error

esp_err_t **gptimer_start** (*gptimer_handle_t* timer)

Start GPTimer (internal counter starts counting)

Note: This function will transit the timer state from "enable" to "run".

Note: This function is allowed to run within ISR context

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

Parameters **`timer`** –**[in]** Timer handle created by `gptimer_new_timer()`
Returns

- ESP_OK: Start GPTimer successfully
- ESP_ERR_INVALID_ARG: Start GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Start GPTimer failed because the timer is not enabled or is already in running
- ESP_FAIL: Start GPTimer failed because of other error

esp_err_t **`gptimer_stop`**(*gptimer_handle_t* timer)

Stop GPTimer (internal counter stops counting)

Note: This function will transit the timer state from "run" to "enable".

Note: This function is allowed to run within ISR context

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

Parameters **`timer`** –**[in]** Timer handle created by `gptimer_new_timer()`
Returns

- ESP_OK: Stop GPTimer successfully
- ESP_ERR_INVALID_ARG: Stop GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Stop GPTimer failed because the timer is not in running.
- ESP_FAIL: Stop GPTimer failed because of other error

Structures

struct **`gptimer_alarm_event_data_t`**

GPTimer alarm event data.

Public Members

uint64_t **`count_value`**

Current count value

uint64_t **`alarm_value`**

Current alarm value

struct **gptimer_event_callbacks_t**

Group of supported GPTimer callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_GPTIMER_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM.

Public Members

gptimer_alarm_cb_t **on_alarm**

Timer alarm callback

struct **gptimer_config_t**

General Purpose Timer configuration.

Public Members

gptimer_clock_source_t **clk_src**

GPTimer clock source

gptimer_count_direction_t **direction**

Count direction

uint32_t **resolution_hz**

Counter resolution (working frequency) in Hz, hence, the step size of each count tick equals to (1 / resolution_hz) seconds

int **intr_priority**

GPTimer interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **intr_shared**

Set true, the timer interrupt number can be shared with other peripherals

struct *gptimer_config_t*::[anonymous] **flags**

GPTimer config flags

struct **gptimer_alarm_config_t**

General Purpose Timer alarm configuration.

Public Members

uint64_t **alarm_count**

Alarm target count value

uint64_t **reload_count**

Alarm reload count value, effect only when `auto_reload_on_alarm` is set to true

uint32_t **auto_reload_on_alarm**

Reload the count value by hardware, immediately at the alarm event

struct *gptimer_alarm_config_t*::[anonymous] **flags**

Alarm config flags

Type Definitions

typedef struct gptimer_t ***gptimer_handle_t**

Type of General Purpose Timer handle.

typedef bool (***gptimer_alarm_cb_t**)(*gptimer_handle_t* timer, const *gptimer_alarm_event_data_t* *edata, void *user_ctx)

Timer alarm callback prototype.

Param timer **[in]** Timer handle created by `gptimer_new_timer()`
Param edata **[in]** Alarm event data, fed by driver
Param user_ctx **[in]** User data, passed from `gptimer_register_event_callbacks()`
Return Whether a high priority task has been waken up by this function

Header File

- components/hal/include/hal/timer_types.h

Type Definitions

typedef *soc_periph_gptimer_clk_src_t* **gptimer_clock_source_t**

GPTimer clock source.

Note: User should select the clock source based on the power and resolution requirement

Enumerations

enum **gptimer_count_direction_t**

GPTimer count direction.

Values:

enumerator **GPTIMER_COUNT_DOWN**

Decrease count value

enumerator **GPTIMER_COUNT_UP**

Increase count value

2.6.8 Inter-Integrated Circuit (I2C)

Overview

I2C is a serial, synchronous, half-duplex communication protocol that allows co-existence of multiple masters and slaves on the same bus. The I2C bus consists of two lines: serial data line (SDA) and serial clock (SCL). Both lines require pull-up resistors.

With such advantages as simplicity and low manufacturing cost, I2C is mostly used for communication of low-speed peripheral devices over short distances (within one foot).

ESP32 has 2 I2C controller (also referred to as port), responsible for handling communications on the I2C bus. A single I2C controller can operate as master or slave.

Driver Features

I2C driver governs communications of devices over the I2C bus. The driver supports the following features:

- Reading and writing bytes in Master mode
- Slave mode
- Reading and writing to registers which are in turn read/written by the master

Driver Usage

The following sections describe typical steps of configuring and operating the I2C driver:

1. *Configuration* - set the initialization parameters (master or slave mode, GPIO pins for SDA and SCL, clock speed, etc.)
2. *Install Driver*- activate the driver on one of the two I2C controllers as a master or slave
3. Depending on whether you configure the driver for a master or slave, choose the appropriate item
    a) *Communication as Master* - handle communications (master)
    b) *Communication as Slave* - respond to messages from the master (slave)
4. *Interrupt Handling* - configure and service I2C interrupts
5. *Customized Configuration* - adjust default I2C communication parameters (timings, bit order, etc.)
6. *Error Handling* - how to recognize and handle driver configuration and communication errors
7. *Delete Driver*- release resources used by the I2C driver when communication ends

Configuration To establish I2C communication, start by configuring the driver. This is done by setting the parameters of the structure `i2c_config_t`:

- Set I2C **mode of operation** - master or slave from `i2c_mode_t`
- Configure **communication pins**
  - Assign GPIO pins for SDA and SCL signals
  - Set whether to enable ESP32' s internal pull-ups
- (Master only) Set I2C **clock speed**
- (Slave only) Configure the following
  - Whether to enable **10 bit address mode**
  - Define **slave address**

After that, initialize the configuration for a given I2C port. For this, call the function `i2c_param_config()` and pass to it the port number and the structure `i2c_config_t`.

Configuration example (master):

```
int i2c_master_port = 0;
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,          // select GPIO specific to your␣
↪project
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
```

(continues on next page)

(continued from previous page)

```
    .scl_io_num = I2C_MASTER_SCL_IO,          // select GPIO specific to your␣
↪project
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,  // select frequency specific to your␣
↪project
    .clk_flags = 0,                           // you can use I2C_SCLK_SRC_FLAG_*␣
↪flags to choose i2c source clock here
};
```

Configuration example (slave):

```
int i2c_slave_port = I2C_SLAVE_NUM;
i2c_config_t conf_slave = {
    .sda_io_num = I2C_SLAVE_SDA_IO,           // select GPIO specific to your␣
↪project
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_SLAVE_SCL_IO,           // select GPIO specific to your␣
↪project
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .mode = I2C_MODE_SLAVE,
    .slave.addr_10bit_en = 0,
    .slave.slave_addr = ESP_SLAVE_ADDR,       // address of your project
    .clk_flags = 0,
};
```

At this stage, *i2c_param_config()* also sets a few other I2C configuration parameters to default values that are defined by the I2C specification. For more details on the values and how to modify them, see *Customized Configuration*.

Source Clock Configuration **Clock sources allocator** is added for supporting different clock sources. The clock allocator will choose one clock source that meets all the requirements of frequency and capability (as requested in *i2c_config_t::clk_flags*).

When *i2c_config_t::clk_flags* is 0, the clock allocator will select only according to the desired frequency. If no special capabilities are needed, such as APB, you can configure the clock allocator to select the source clock only according to the desired frequency. For this, set *i2c_config_t::clk_flags* to 0. For clock characteristics, see the table below.

Note: A clock is not a valid option, if it doesn't meet the requested capabilities, i.e. any bit of requested capabilities (clk_flags) is 0 in the clock's capabilities.

Table 3: Characteristics of ESP32 clock sources

Clock name	Clock frequency	MAX freq for SCL	Clock capabilities
APB clock	80 MHz	4 MHz	/

Explanations for *i2c_config_t::clk_flags* are as follows:

1. *I2C_SCLK_SRC_FLAG_AWARE_DFS*: Clock's baud rate will not change while APB clock is changing.
2. *I2C_SCLK_SRC_FLAG_LIGHT_SLEEP*: It supports Light-sleep mode, which APB clock cannot do.
3. Some flags may not be supported on ESP32, reading technical reference manual before using it.

Note: The clock frequency of SCL in master mode should not be lager than max frequency for SCL mentioned in the table above.

Note: The clock frequency of SCL will be influenced by the pull-up resistors and wire capacitance (or might slave capacitance) together. Therefore, users need to choose correct pull-up resistors by themselves to make the frequency accurate. It is recommended by I2C protocol that the pull-up resistors commonly range from 1KOhms to 10KOhms, but different frequencies need different resistors.

Generally speaking, the higher frequency is selected, the smaller resistor should be used (but not less than 1KOhms). This is because high resistor will decline the current, which will lengthen the rising time and reduce the frequency. Usually, range 2KOhms to 5KOhms is what we recommend, but users also might need to make some adjustment depends on their reality.

Install Driver After the I2C driver is configured, install it by calling the function `i2c_driver_install()` with the following parameters:

- Port number, one of the two port numbers from `i2c_port_t`
- master or slave, selected from `i2c_mode_t`
- (Slave only) Size of buffers to allocate for sending and receiving data. As I2C is a master-centric bus, data can only go from the slave to the master at the master's request. Therefore, the slave will usually have a send buffer where the slave application writes data. The data remains in the send buffer to be read by the master at the master's own discretion.
- Flags for allocating the interrupt (see ESP_INTR_FLAG_* values in esp_hw_support/include/esp_intr_alloc.h)

Communication as Master After installing the I2C driver, ESP32 is ready to communicate with other I2C devices.

ESP32's I2C controller operating as master is responsible for establishing communication with I2C slave devices and sending commands to trigger a slave to action, for example, to take a measurement and send the readings back to the master.

For better process organization, the driver provides a container, called a "command link", that should be populated with a sequence of commands and then passed to the I2C controller for execution.

Master Write The example below shows how to build a command link for an I2C master to send *n* bytes to a slave.

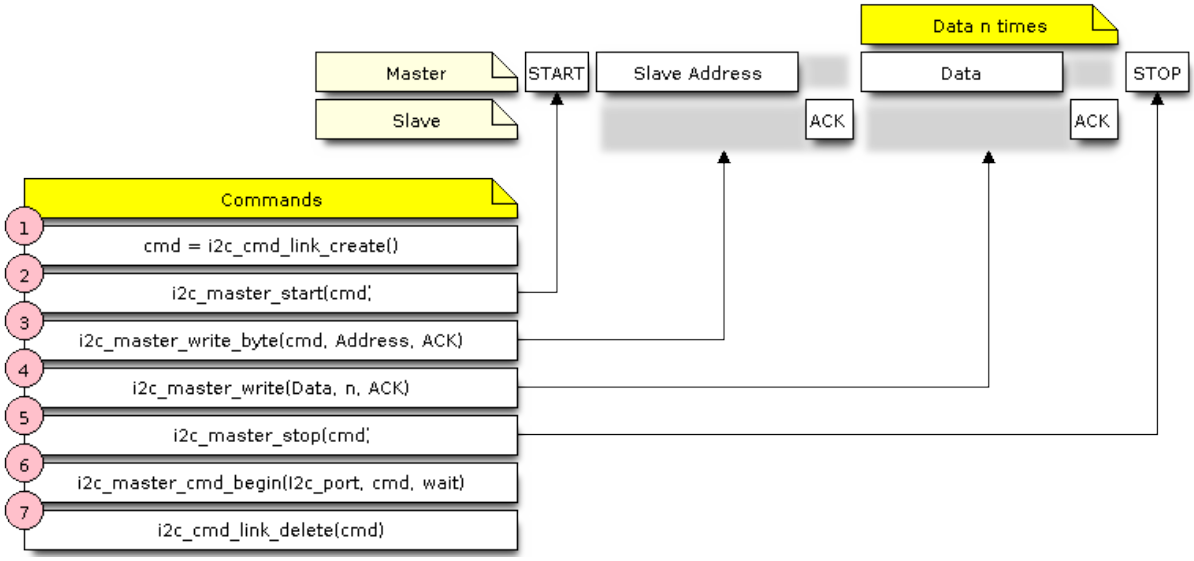


Fig. 7: I2C command link - master write example

The following describes how a command link for a "master write" is set up and what comes inside:

1. Create a command link with *i2c_cmd_link_create()*.
   Then, populate it with the series of data to be sent to the slave:
   a) **Start bit** - *i2c_master_start()*
   b) **Slave address** - *i2c_master_write_byte()*. The single byte address is provided as an argument of this function call.
   c) **Data** - One or more bytes as an argument of *i2c_master_write()*
   d) **Stop bit** - *i2c_master_stop()*
   Both functions *i2c_master_write_byte()* and *i2c_master_write()* have an additional argument specifying whether the master should ensure that it has received the ACK bit.
2. Trigger the execution of the command link by I2C controller by calling *i2c_master_cmd_begin()*. Once the execution is triggered, the command link cannot be modified.
3. After the commands are transmitted, release the resources used by the command link by calling *i2c_cmd_link_delete()*.

Master Read The example below shows how to build a command link for an I2C master to read *n* bytes from a slave.

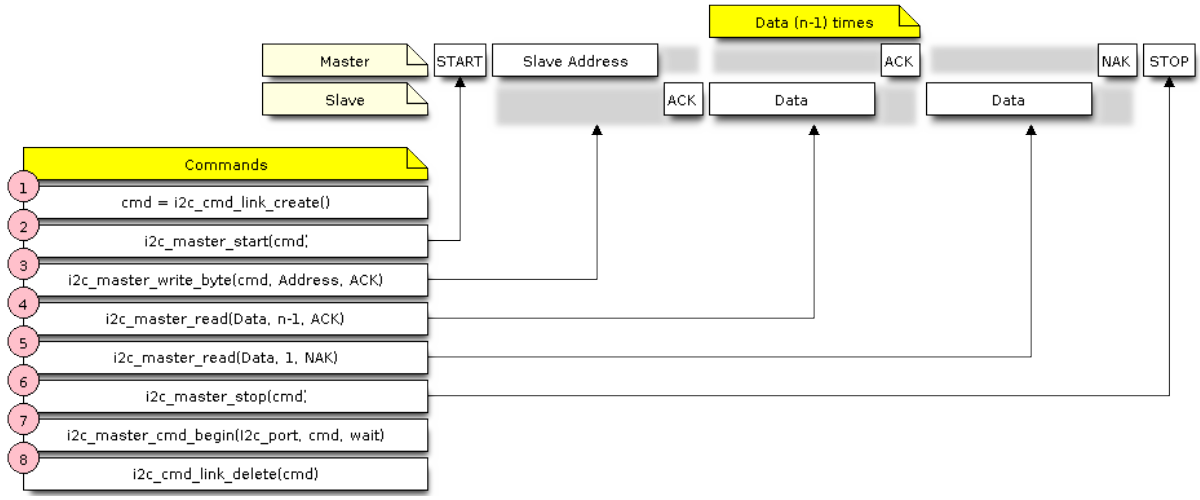


Fig. 8: I2C command link - master read example

Compared to writing data, the command link is populated in Step 4 not with `i2c_master_write...` functions but with *i2c_master_read_byte()* and / or *i2c_master_read()*. Also, the last read in Step 5 is configured so that the master does not provide the ACK bit.

Indicating Write or Read After sending a slave address (see Step 3 on both diagrams above), the master either writes or reads from the slave.

The information on what the master will actually do is hidden in the least significant bit of the slave's address.

For this reason, the command link sent by the master to write data to the slave contains the address `(ESP_SLAVE_ADDR << 1) | I2C_MASTER_WRITE` and looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_WRITE, ACK_EN);
```

Likewise, the command link to read from the slave looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_READ, ACK_EN);
```

Communication as Slave After installing the I2C driver, ESP32 is ready to communicate with other I2C devices.

The API provides the following functions for slaves

- *i2c_slave_read_buffer()*

Whenever the master writes data to the slave, the slave will automatically store it in the receive buffer. This allows the slave application to call the function `i2c_slave_read_buffer()` at its own discretion. This function also has a parameter to specify block time if no data is in the receive buffer. This will allow the slave application to wait with a specified timeout for data to arrive to the buffer.

- `i2c_slave_write_buffer()`
  The send buffer is used to store all the data that the slave wants to send to the master in FIFO order. The data stays there until the master requests for it. The function `i2c_slave_write_buffer()` has a parameter to specify block time if the send buffer is full. This will allow the slave application to wait with a specified timeout for the adequate amount of space to become available in the send buffer.

A code example showing how to use these functions can be found in peripherals/i2c.

Interrupt Handling During driver installation, an interrupt handler is installed by default.

Customized Configuration As mentioned at the end of Section *Configuration*, when the function `i2c_param_config()` initializes the driver configuration for an I2C port, it also sets several I2C communication parameters to default values defined in the I2C specification. Some other related parameters are pre-configured in registers of the I2C controller.

All these parameters can be changed to user-defined values by calling dedicated functions given in the table below. Please note that the timing values are defined in APB clock cycles. The frequency of APB is specified in `I2C_APB_CLK_FREQ`.

Table 4: Other Configurable I2C Communication Parameters

Parameters to Change	Function
High time and low time for SCL pulses	`i2c_set_period()`
SCL and SDA signal timing used during generation of **start** signals	`i2c_set_start_timing()`
SCL and SDA signal timing used during generation of **stop** signals	`i2c_set_stop_timing()`
Timing relationship between SCL and SDA signals when slave samples, as well as when master toggles	`i2c_set_data_timing()`
I2C timeout	`i2c_set_timeout()`
Choice between transmitting / receiving the LSB or MSB first, choose one of the modes defined in `i2c_trans_mode_t`	`i2c_set_data_mode()`

Each of the above functions has a *_get_* counterpart to check the currently set value. For example, to check the I2C timeout value, call `i2c_get_timeout()`.

To check the default parameter values which are set during the driver configuration process, please refer to the file driver/i2c.c and look for defines with the suffix `_DEFAULT`.

You can also select different pins for SDA and SCL signals and alter the configuration of pull-ups with the function `i2c_set_pin()`. If you want to modify already entered values, use the function `i2c_param_config()`.

Note: ESP32's internal pull-ups are in the range of tens of kOhm, which is, in most cases, insufficient for use as I2C pull-ups. Users are advised to use external pull-ups with values described in the I2C specification. For help with calculating the resistor values see TI Application Note

---

Error Handling The majority of I2C driver functions either return `ESP_OK` on successful completion or a specific error code on failure. It is a good practice to always check the returned values and implement error handling. The driver also prints out log messages that contain error details, e.g., when checking the validity of entered configuration. For details please refer to the file driver/i2c.c and look for defines with the suffix `_ERR_STR`.

Use dedicated interrupts to capture communication failures. For instance, if a slave stretches the clock for too long while preparing the data to send back to master, the interrupt `I2C_TIME_OUT_INT` will be triggered. For detailed information, see *Interrupt Handling*.

In case of a communication failure, you can reset the internal hardware buffers by calling the functions `i2c_reset_tx_fifo()` and `i2c_reset_rx_fifo()` for the send and receive buffers respectively.

Delete Driver When the I2C communication is established with the function `i2c_driver_install()` and is not required for some substantial amount of time, the driver may be deinitialized to release allocated resources by calling `i2c_driver_delete()`.

Before calling `i2c_driver_delete()` to remove i2c driver, please make sure that all threads have stopped using the driver in any way, because this function does not guarantee thread safety.

Application Example

I2C examples: peripherals/i2c.

API Reference

Header File

- components/driver/include/driver/i2c.h

Functions

esp_err_t **i2c_driver_install** (*i2c_port_t* i2c_num, *i2c_mode_t* mode, size_t slv_rx_buf_len, size_t slv_tx_buf_len, int intr_alloc_flags)

Install an I2C driver.

Note: Not all Espressif chips can support slave mode (e.g. ESP32C2)

Note: In master mode, if the cache is likely to be disabled(such as write flash) and the slave is time-sensitive, `ESP_INTR_FLAG_IRAM` is suggested to be used. In this case, please use the memory allocated from internal RAM in i2c read and write function, because we can not access the psram(if psram is enabled) in interrupt handle function when cache is disabled.

Parameters

- **i2c_num** –I2C port number
- **mode** –I2C mode (either master or slave).
- **slv_rx_buf_len** –Receiving buffer size. Only slave mode will use this value, it is ignored in master mode.
- **slv_tx_buf_len** –Sending buffer size. Only slave mode will use this value, it is ignored in master mode.
- **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver installation error

esp_err_t **i2c_driver_delete**(*i2c_port_t* i2c_num)

Delete I2C driver.

Note: This function does not guarantee thread safety. Please make sure that no thread will continuously hold semaphores before calling the delete function.

Parameters **i2c_num** –I2C port to delete

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_param_config**(*i2c_port_t* i2c_num, const *i2c_config_t* *i2c_conf)

Configure an I2C bus with the given configuration.

Parameters

- **i2c_num** –I2C port to configure
- **i2c_conf** –Pointer to the I2C configuration

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_reset_tx_fifo**(*i2c_port_t* i2c_num)

reset I2C tx hardware fifo

Parameters **i2c_num** –I2C port number

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_reset_rx_fifo**(*i2c_port_t* i2c_num)

reset I2C rx fifo

Parameters **i2c_num** –I2C port number

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_pin**(*i2c_port_t* i2c_num, int sda_io_num, int scl_io_num, bool sda_pullup_en, bool scl_pullup_en, *i2c_mode_t* mode)

Configure GPIO pins for I2C SCK and SDA signals.

Parameters

- **i2c_num** –I2C port number
- **sda_io_num** –GPIO number for I2C SDA signal
- **scl_io_num** –GPIO number for I2C SCL signal
- **sda_pullup_en** –Enable the internal pullup for SDA pin
- **scl_pullup_en** –Enable the internal pullup for SCL pin
- **mode** –I2C mode

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_master_write_to_device**(*i2c_port_t* i2c_num, uint8_t device_address, const uint8_t *write_buffer, size_t write_size, TickType_t ticks_to_wait)

Perform a write to a device connected to a particular I2C port. This function is a wrapper to `i2c_master_start()`, `i2c_master_write()`, `i2c_master_read()`, etc···It shall only be called in I2C master mode.

Parameters

- **i2c_num** –I2C port number to perform the transfer on

- **device_address** –I2C device’ s 7-bit address
- **write_buffer** –Bytes to send on the bus
- **write_size** –Size, in bytes, of the write buffer
- **ticks_to_wait** –Maximum ticks to wait before issuing a timeout.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave hasn’ t ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_master_read_from_device**(*i2c_port_t* i2c_num, uint8_t device_address, uint8_t *read_buffer, size_t read_size, TickType_t ticks_to_wait)

Perform a read to a device connected to a particular I2C port. This function is a wrapper to `i2c_master_start()`, `i2c_master_write()`, `i2c_master_read()`, etc··· It shall only be called in I2C master mode.

Parameters

- **i2c_num** –I2C port number to perform the transfer on
- **device_address** –I2C device’ s 7-bit address
- **read_buffer** –Buffer to store the bytes received on the bus
- **read_size** –Size, in bytes, of the read buffer
- **ticks_to_wait** –Maximum ticks to wait before issuing a timeout.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave hasn’ t ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_master_write_read_device**(*i2c_port_t* i2c_num, uint8_t device_address, const uint8_t *write_buffer, size_t write_size, uint8_t *read_buffer, size_t read_size, TickType_t ticks_to_wait)

Perform a write followed by a read to a device on the I2C bus. A repeated start signal is used between the `write` and `read`, thus, the bus is not released until the two transactions are finished. This function is a wrapper to `i2c_master_start()`, `i2c_master_write()`, `i2c_master_read()`, etc··· It shall only be called in I2C master mode.

Parameters

- **i2c_num** –I2C port number to perform the transfer on
- **device_address** –I2C device’ s 7-bit address
- **write_buffer** –Bytes to send on the bus
- **write_size** –Size, in bytes, of the write buffer
- **read_buffer** –Buffer to store the bytes received on the bus
- **read_size** –Size, in bytes, of the read buffer
- **ticks_to_wait** –Maximum ticks to wait before issuing a timeout.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave hasn’ t ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

i2c_cmd_handle_t **i2c_cmd_link_create_static**(uint8_t *buffer, uint32_t size)

Create and initialize an I2C commands list with a given buffer. All the allocations for data or signals (START, STOP, ACK, ···) will be performed within this buffer. This buffer must be valid during the whole transaction. After finishing the I2C transactions, it is required to call `i2c_cmd_link_delete_static()`.

Note: It is **highly** advised to not allocate this buffer on the stack. The size of the data used underneath may

increase in the future, resulting in a possible stack overflow as the macro `I2C_LINK_RECOMMENDED_SIZE` would also return a bigger value. A better option is to use a buffer allocated statically or dynamically (with `malloc`).

Parameters

- **`buffer`** –Buffer to use for commands allocations
- **`size`** –Size in bytes of the buffer

Returns Handle to the I2C command link or NULL if the buffer provided is too small, please use `I2C_LINK_RECOMMENDED_SIZE` macro to get the recommended size for the buffer.

i2c_cmd_handle_t **`i2c_cmd_link_create`**(void)

Create and initialize an I2C commands list with a given buffer. After finishing the I2C transactions, it is required to call `i2c_cmd_link_delete()` to release and return the resources. The required bytes will be dynamically allocated.

Returns Handle to the I2C command link or NULL in case of insufficient dynamic memory.

void **`i2c_cmd_link_delete_static`**(*i2c_cmd_handle_t* cmd_handle)

Free the I2C commands list allocated statically with `i2c_cmd_link_create_static`.

Parameters **`cmd_handle`** –I2C commands list allocated statically. This handle should be created thanks to `i2c_cmd_link_create_static()` function

void **`i2c_cmd_link_delete`**(*i2c_cmd_handle_t* cmd_handle)

Free the I2C commands list.

Parameters **`cmd_handle`** –I2C commands list. This handle should be created thanks to `i2c_cmd_link_create()` function

esp_err_t **`i2c_master_start`**(*i2c_cmd_handle_t* cmd_handle)

Queue a "START signal" to the given commands list. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all the queued commands.

Parameters **`cmd_handle`** –I2C commands list

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **`i2c_master_write_byte`**(*i2c_cmd_handle_t* cmd_handle, uint8_t data, bool ack_en)

Queue a "write byte" command to the commands list. A single byte will be sent on the I2C port. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all queued commands.

Parameters

- **`cmd_handle`** –I2C commands list
- **`data`** –Byte to send on the port
- **`ack_en`** –Enable ACK signal

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **`i2c_master_write`**(*i2c_cmd_handle_t* cmd_handle, const uint8_t *data, size_t data_len, bool ack_en)

Queue a "write (multiple) bytes" command to the commands list. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all queued commands.

Parameters

- **`cmd_handle`** –I2C commands list

- **data** –Bytes to send. This buffer shall remain **valid** until the transaction is finished. If the PSRAM is enabled and `intr_flag` is set to `ESP_INTR_FLAG_IRAM`, `data` should be allocated from internal RAM.
- **data_len** –Length, in bytes, of the data buffer
- **ack_en** –Enable ACK signal

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **i2c_master_read_byte**(*i2c_cmd_handle_t* cmd_handle, uint8_t *data, *i2c_ack_type_t* ack)

Queue a "read byte" command to the commands list. A single byte will be read on the I2C bus. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all queued commands.

Parameters

- **cmd_handle** –I2C commands list
- **data** –Pointer where the received byte will the stored. This buffer shall remain **valid** until the transaction is finished.
- **ack** –ACK signal

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **i2c_master_read**(*i2c_cmd_handle_t* cmd_handle, uint8_t *data, size_t data_len, *i2c_ack_type_t* ack)

Queue a "read (multiple) bytes" command to the commands list. Multiple bytes will be read on the I2C bus. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all queued commands.

Parameters

- **cmd_handle** –I2C commands list
- **data** –Pointer where the received bytes will the stored. This buffer shall remain **valid** until the transaction is finished.
- **data_len** –Size, in bytes, of the `data` buffer
- **ack** –ACK signal

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **i2c_master_stop**(*i2c_cmd_handle_t* cmd_handle)

Queue a "STOP signal" to the given commands list. This function shall only be called in I2C master mode. Call `i2c_master_cmd_begin()` to send all the queued commands.

Parameters **cmd_handle** –I2C commands list

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM The static buffer used to create `cmd_handler` is too small
- ESP_FAIL No more memory left on the heap

esp_err_t **i2c_master_cmd_begin**(*i2c_port_t* i2c_num, *i2c_cmd_handle_t* cmd_handle, TickType_t ticks_to_wait)

Send all the queued commands on the I2C bus, in master mode. The task will be blocked until all the commands have been sent out. The I2C port is protected by mutex, so this function is thread-safe. This function shall only be called in I2C master mode.

Parameters

- **i2c_num** –I2C port number
- **cmd_handle** –I2C commands list
- **ticks_to_wait** –Maximum ticks to wait before issuing a timeout.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave hasn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

int **i2c_slave_write_buffer**(*i2c_port_t* i2c_num, const uint8_t *data, int size, TickType_t ticks_to_wait)

Write bytes to internal ringbuffer of the I2C slave data. When the TX fifo empty, the ISR will fill the hardware FIFO with the internal ringbuffer's data.

Note: This function shall only be called in I2C slave mode.

Parameters

- **i2c_num** –I2C port number
- **data** –Bytes to write into internal buffer
- **size** –Size, in bytes, of `data` buffer
- **ticks_to_wait** –Maximum ticks to wait.

Returns

- ESP_FAIL (-1) Parameter error
- Other (>=0) The number of data bytes pushed to the I2C slave buffer.

int **i2c_slave_read_buffer**(*i2c_port_t* i2c_num, uint8_t *data, size_t max_size, TickType_t ticks_to_wait)

Read bytes from I2C internal buffer. When the I2C bus receives data, the ISR will copy them from the hardware RX FIFO to the internal ringbuffer. Calling this function will then copy bytes from the internal ringbuffer to the `data` user buffer.

Note: This function shall only be called in I2C slave mode.

Parameters

- **i2c_num** –I2C port number
- **data** –Buffer to fill with ringbuffer's bytes
- **max_size** –Maximum bytes to read
- **ticks_to_wait** –Maximum waiting ticks

Returns

- ESP_FAIL(-1) Parameter error
- Others(>=0) The number of data bytes read from I2C slave buffer.

esp_err_t **i2c_set_period**(*i2c_port_t* i2c_num, int high_period, int low_period)

Set I2C master clock period.

Parameters

- **i2c_num** –I2C port number
- **high_period** –Clock cycle number during SCL is high level, high_period is a 14 bit value
- **low_period** –Clock cycle number during SCL is low level, low_period is a 14 bit value

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_period**(*i2c_port_t* i2c_num, int *high_period, int *low_period)

Get I2C master clock period.

Parameters

- **i2c_num** –I2C port number
- **high_period** –pointer to get clock cycle number during SCL is high level, will get a 14 bit value
- **low_period** –pointer to get clock cycle number during SCL is low level, will get a 14 bit value

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_filter_enable**(*i2c_port_t* i2c_num, uint8_t cyc_num)

Enable hardware filter on I2C bus Sometimes the I2C bus is disturbed by high frequency noise(about 20ns), or the rising edge of the SCL clock is very slow, these may cause the master state machine to break. Enable hardware filter can filter out high frequency interference and make the master more stable.

Note: Enable filter will slow down the SCL clock.

Parameters

- **i2c_num** –I2C port number to filter
- **cyc_num** –the APB cycles need to be filtered (0<= cyc_num <=7). When the period of a pulse is less than cyc_num * APB_cycle, the I2C controller will ignore this pulse.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_filter_disable**(*i2c_port_t* i2c_num)

Disable filter on I2C bus.

Parameters **i2c_num** –I2C port number

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_start_timing**(*i2c_port_t* i2c_num, int setup_time, int hold_time)

set I2C master start signal timing

Parameters

- **i2c_num** –I2C port number
- **setup_time** –clock number between the falling-edge of SDA and rising-edge of SCL for start mark, it’ s a 10-bit value.
- **hold_time** –clock num between the falling-edge of SDA and falling-edge of SCL for start mark, it’ s a 10-bit value.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_start_timing**(*i2c_port_t* i2c_num, int *setup_time, int *hold_time)

get I2C master start signal timing

Parameters

- **i2c_num** –I2C port number
- **setup_time** –pointer to get setup time
- **hold_time** –pointer to get hold time

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_stop_timing**(*i2c_port_t* i2c_num, int setup_time, int hold_time)

set I2C master stop signal timing

Parameters
- **i2c_num** –I2C port number
- **setup_time** –clock num between the rising-edge of SCL and the rising-edge of SDA, it's a 10-bit value.
- **hold_time** –clock number after the STOP bit's rising-edge, it's a 14-bit value.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_stop_timing**(*i2c_port_t* i2c_num, int *setup_time, int *hold_time)

get I2C master stop signal timing

Parameters
- **i2c_num** –I2C port number
- **setup_time** –pointer to get setup time.
- **hold_time** –pointer to get hold time.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_data_timing**(*i2c_port_t* i2c_num, int sample_time, int hold_time)

set I2C data signal timing

Parameters
- **i2c_num** –I2C port number
- **sample_time** –clock number I2C used to sample data on SDA after the rising-edge of SCL, it's a 10-bit value
- **hold_time** –clock number I2C used to hold the data after the falling-edge of SCL, it's a 10-bit value

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_data_timing**(*i2c_port_t* i2c_num, int *sample_time, int *hold_time)

get I2C data signal timing

Parameters
- **i2c_num** –I2C port number
- **sample_time** –pointer to get sample time
- **hold_time** –pointer to get hold time

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_timeout**(*i2c_port_t* i2c_num, int timeout)

set I2C timeout value

Parameters
- **i2c_num** –I2C port number
- **timeout** –timeout value for I2C bus (unit: APB 80Mhz clock cycle)

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_timeout**(*i2c_port_t* i2c_num, int *timeout)

get I2C timeout value

Parameters
- **i2c_num** –I2C port number

- **timeout** –pointer to get timeout value

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_set_data_mode**(*i2c_port_t* i2c_num, *i2c_trans_mode_t* tx_trans_mode, *i2c_trans_mode_t* rx_trans_mode)

set I2C data transfer mode

Parameters

- **i2c_num** –I2C port number
- **tx_trans_mode** –I2C sending data mode
- **rx_trans_mode** –I2C receving data mode

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2c_get_data_mode**(*i2c_port_t* i2c_num, *i2c_trans_mode_t* *tx_trans_mode, *i2c_trans_mode_t* *rx_trans_mode)

get I2C data transfer mode

Parameters

- **i2c_num** –I2C port number
- **tx_trans_mode** –pointer to get I2C sending data mode
- **rx_trans_mode** –pointer to get I2C receiving data mode

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Structures

struct **i2c_config_t**

I2C initialization parameters.

Public Members

i2c_mode_t **mode**

I2C mode

int **sda_io_num**

GPIO number for I2C sda signal

int **scl_io_num**

GPIO number for I2C scl signal

bool **sda_pullup_en**

Internal GPIO pull mode for I2C sda signal

bool **scl_pullup_en**

Internal GPIO pull mode for I2C scl signal

uint32_t **clk_speed**

I2C clock frequency for master mode, (no higher than 1MHz for now)

struct *i2c_config_t*::[anonymous]::[anonymous] **master**

I2C master config

uint8_t **addr_10bit_en**

I2C 10bit address mode enable for slave mode

uint16_t **slave_addr**

I2C address for slave mode

uint32_t **maximum_speed**

I2C expected clock speed from SCL.

struct *i2c_config_t*::[anonymous]::[anonymous] **slave**

I2C slave config

uint32_t **clk_flags**

Bitwise of `I2C_SCLK_SRC_FLAG_**FOR_DFS**` for clk source choice

Macros

I2C_APB_CLK_FREQ

I2C source clock is APB clock, 80MHz

I2C_NUM_MAX

I2C port max

I2C_NUM_0

I2C port 0

I2C_NUM_1

I2C port 1

I2C_SCLK_SRC_FLAG_FOR_NOMAL

Any one clock source that is available for the specified frequency may be choosen

I2C_SCLK_SRC_FLAG_AWARE_DFS

For REF tick clock, it won't change with APB.

I2C_SCLK_SRC_FLAG_LIGHT_SLEEP

For light sleep mode.

I2C_INTERNAL_STRUCT_SIZE

Minimum size, in bytes, of the internal private structure used to describe I2C commands link.

I2C_LINK_RECOMMENDED_SIZE(TRANSACTIONS)

The following macro is used to determine the recommended size of the buffer to pass to `i2c_cmd_link_create_static()` function. It requires one parameter, `TRANSACTIONS`, describing the number of transactions intended to be performed on the I2C port. For example, if one wants to perform a read on an I2C device register, `TRANSACTIONS` must be at least 2, because the commands required are the following:

- write device register
- read register content

Signals such as "(repeated) start", "stop", "nack", "ack" shall not be counted.

Type Definitions

typedef void ***i2c_cmd_handle_t**

I2C command handle

Header File

- components/hal/include/hal/i2c_types.h

Type Definitions

typedef int **i2c_port_t**

I2C port number, can be I2C_NUM_0 ~ (I2C_NUM_MAX-1).

typedef *soc_periph_i2c_clk_src_t* **i2c_clock_source_t**

I2C group clock source.

Enumerations

enum **i2c_mode_t**

Values:

enumerator **I2C_MODE_SLAVE**

I2C slave mode

enumerator **I2C_MODE_MASTER**

I2C master mode

enumerator **I2C_MODE_MAX**

enum **i2c_rw_t**

Values:

enumerator **I2C_MASTER_WRITE**

I2C write data

enumerator **I2C_MASTER_READ**

I2C read data

enum **i2c_trans_mode_t**

Values:

enumerator **I2C_DATA_MODE_MSB_FIRST**

I2C data msb first

enumerator **I2C_DATA_MODE_LSB_FIRST**

I2C data lsb first

enumerator **I2C_DATA_MODE_MAX**

enum **i2c_addr_mode_t**

Values:

enumerator **I2C_ADDR_BIT_7**

I2C 7bit address for slave mode

enumerator **I2C_ADDR_BIT_10**

I2C 10bit address for slave mode

enumerator **I2C_ADDR_BIT_MAX**

enum **i2c_ack_type_t**

Values:

enumerator **I2C_MASTER_ACK**

I2C ack for each byte read

enumerator **I2C_MASTER_NACK**

I2C nack for each byte read

enumerator **I2C_MASTER_LAST_NACK**

I2C nack for the last byte

enumerator **I2C_MASTER_ACK_MAX**

2.6.9 Inter-IC Sound (I2S)

Introduction

I2S (Inter-IC Sound) is a synchronous serial communication protocol usually used for transmitting audio data between two digital audio devices.

ESP32 contains two I2S peripheral(s). These peripherals can be configured to input and output sample data via the I2S driver.

An I2S bus that communicates in standard or TDM mode consists of the following lines:

- **MCLK:** Master clock line. It is an optional signal depending on the slave side, mainly used for offering a reference clock to the I2S slave device.
- **BCLK:** Bit clock line. The bit clock for data line.
- **WS:** Word (Slot) select line. It is usually used to identify the vocal tract except PDM mode.
- **DIN/DOUT:** Serial data input/output line. Data will loopback internally if DIN and DOUT are set to a same GPIO.

An I2S bus that communicates in PDM mode consists of the following lines:

- **CLK:** PDM clock line.
- **DIN/DOUT:** Serial data input/output line.

Each I2S controller has the following features that can be configured by the I2S driver:

- Operation as system master or slave
- Capable of acting as transmitter or receiver
- DMA controller that allows stream sampling of data without requiring the CPU to copy each data sample

Each controller supports single RX or TX simplex communication. As RX and TX channels share a clock, they can only be combined with the same configuration to establish a full-duplex communication.

I2S File Structure

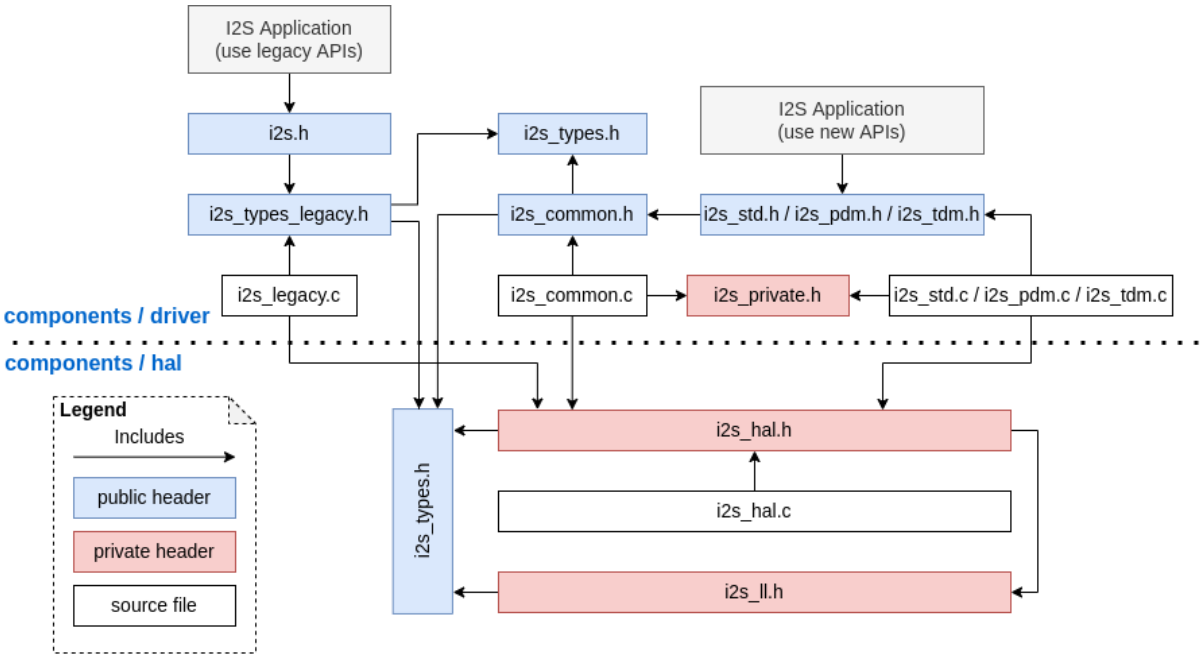


Fig. 9: I2S File Structure

Public headers that need to be included in the I2S application are as follows:

- `i2s.h`: The header file that provides legacy I2S APIs (for apps using legacy driver).
- `i2s_std.h`: The header file that provides standard communication mode specific APIs (for apps using new driver with standard mode).
- `i2s_pdm.h`: The header file that provides PDM communication mode specific APIs (for apps using new driver with PDM mode).
- `i2s_tdm.h`: The header file that provides TDM communication mode specific APIs (for apps using new driver with TDM mode).

Note: The legacy driver cannot coexist with the new driver. Include `i2s.h` to use the legacy driver, or include the other three headers to use the new driver. The legacy driver might be removed in future.

Public headers that have been included in the headers above are as follows:

- `i2s_types_legacy.h`: The header file that provides legacy public types that are only used in the legacy driver.
- `i2s_types.h`: The header file that provides public types.
- `i2s_common.h`: The header file that provides common APIs for all communication modes.

I2S Clock

Clock Source

- `i2s_clock_src_t::I2S_CLK_SRC_DEFAULT`: Default PLL clock.

- `i2s_clock_src_t::I2S_CLK_SRC_PLL_160M`: 160 MHz PLL clock.
- `i2s_clock_src_t::I2S_CLK_SRC_APLL`: Audio PLL clock, which is more precise than `I2S_CLK_SRC_PLL_160M` in high sample rate applications. Its frequency is configurable according to the sample rate. However, if APLL has been occupied by EMAC or other channels, the APLL frequency cannot be changed, and the driver will try to work under this APLL frequency. If this frequency cannot meet the requirements of I2S, the clock configuration will fail.

Clock Terminology

- **Sample rate**: The number of sampled data in one second per slot.
- **SCLK**: Source clock frequency. It is the frequency of the clock source.
- **MCLK**: Master clock frequency. BCLK is generated from this clock. The MCLK signal usually serves as a reference clock and is mostly needed to synchronize BCLK and WS between I2S master and slave roles.
- **BCLK**: Bit clock frequency. Every tick of this clock stands for one data bit on data pin. The slot bit width configured in *`i2s_std_slot_config_t::slot_bit_width`* is equal to the number of BCLK ticks, which means there will be 8/16/24/32 BCLK ticks in one slot.
- **LRCK / WS**: Left/right clock or word select clock. For non-PDM mode, its frequency is equal to the sample rate.

Note: Normally, MCLK should be the multiple of `sample rate` and BCLK at the same time. The field *`i2s_std_clk_config_t::mclk_multiple`* indicates the multiple of MCLK to the `sample rate`. In most cases, `I2S_MCLK_MULTIPLE_256` should be enough. However, if `slot_bit_width` is set to `I2S_SLOT_BIT_WIDTH_24BIT`, to keep MCLK a multiple to the BCLK, *`i2s_std_clk_config_t::mclk_multiple`* should be set to multiples that are divisible by 3 such as `I2S_MCLK_MULTIPLE_384`. Otherwise, WS will be inaccurate.

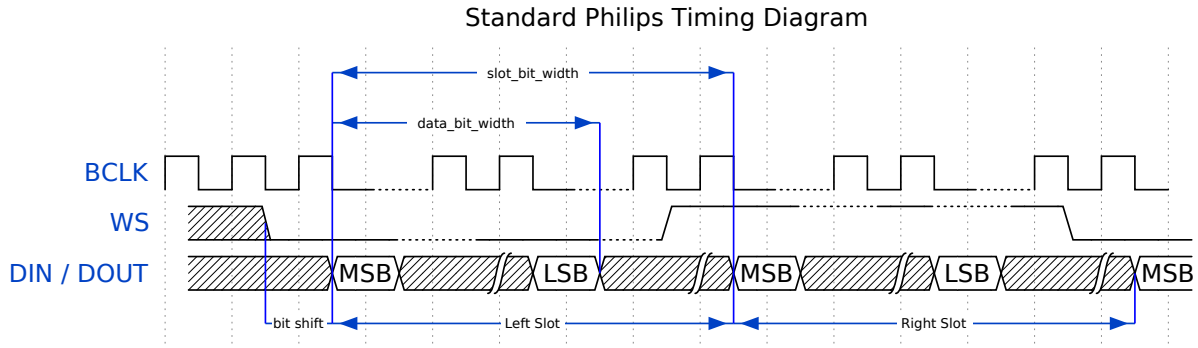
I2S Communication Mode

Overview of All Modes

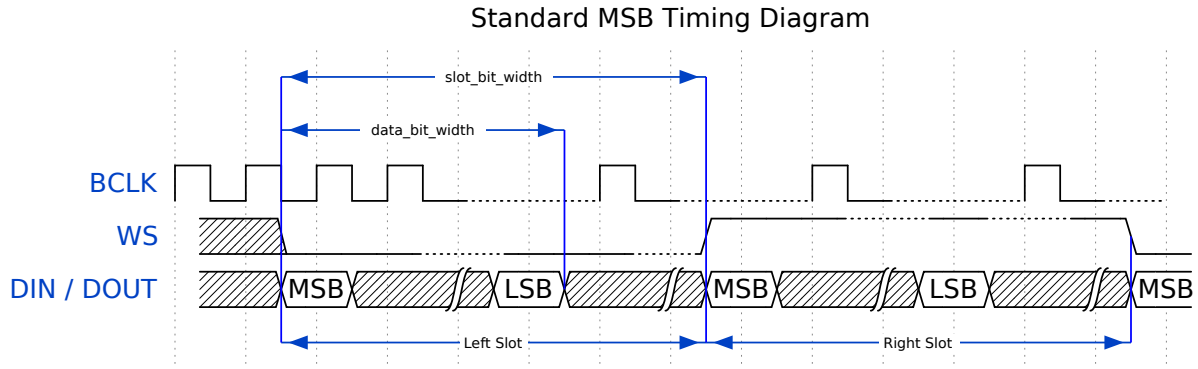
Target	Standard	PDM TX	PDM RX	TDM	ADC/DAC	LCD/Camera
ESP-32	I2S 0/1	I2S 0	I2S 0	none	I2S 0	I2S 0
ESP32-S2	I2S 0	none	none	none	none	I2S 0
ESP32-C3	I2S 0	I2S 0	none	I2S0	none	none
ESP32-S3	I2S 0/1	I2S 0	I2S 0	I2S 0/1	none	none

Standard Mode In standard mode, there are always two sound channels, i.e., the left and right channels, which are called "slots". These slots support 8/16/24/32-bit width sample data. The communication format for the slots mainly includes the followings:

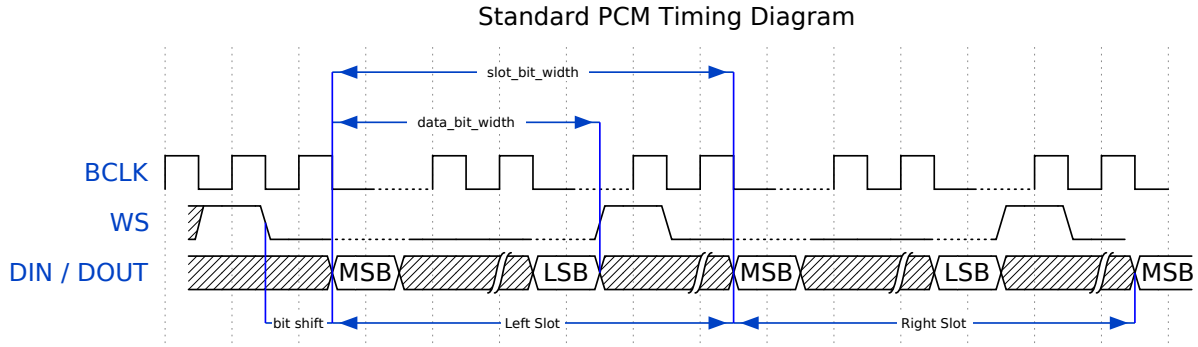
- **Philips Format**: Data signal has one-bit shift comparing to the WS signal, and the duty of WS signal is 50%.



- **MSB Format**: Basically the same as Philips format, but without data shift.

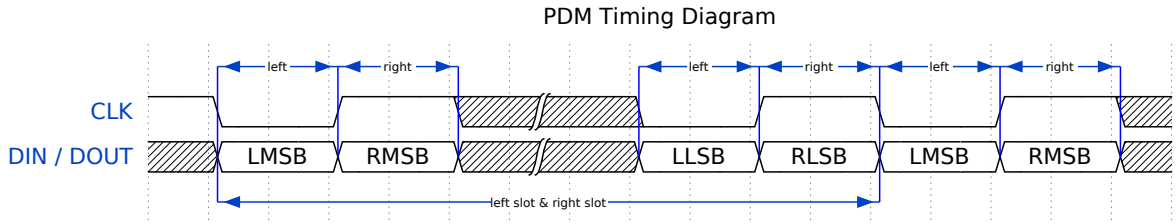


- **PCM Short Format**: Data has one-bit shift and meanwhile the WS signal becomes a pulse lasting for one BCLK cycle.



PDM Mode (TX) PDM (Pulse-density Modulation) mode for the TX channel can convert PCM data into PDM format which always has left and right slots. PDM TX is only supported on I2S0 and it only supports 16-bit width sample data. It needs at least a CLK pin for clock signal and a DOUT pin for data signal (i.e., the WS and SD signal in the following figure; the BCK signal is an internal bit sampling clock, which is not needed between PDM devices). This mode allows users to configure the up-sampling parameters *i2s_pdm_tx_clk_config_t::up_sample_fp* and *i2s_pdm_tx_clk_config_t::up_sample_fs*. The up-sampling rate can be calculated by `up_sample_rate = i2s_pdm_tx_clk_config_t::up_sample_fp / i2s_pdm_tx_clk_config_t::up_sample_fs`. There are two up-sampling modes in PDM TX:

- **Fixed Clock Frequency**: In this mode, the up-sampling rate changes according to the sample rate. Setting `fp = 960` and `fs = sample_rate / 100`, then the clock frequency (Fpdm) on CLK pin will be fixed to `128 * 48 KHz = 6.144 MHz`. Note that this frequency is not equal to the sample rate (Fpcm).
- **Fixed Up-sampling Rate**: In this mode, the up-sampling rate is fixed to 2. Setting `fp = 960` and `fs = 480`, then the clock frequency (Fpdm) on CLK pin will be `128 * sample_rate`.



PDM Mode (RX) PDM (Pulse-density Modulation) mode for RX channel can receive PDM-format data and convert the data into PCM format. PDM RX is only supported on I2S0, and it only supports 16-bit width sample data. PDM RX needs at least a CLK pin for clock signal and a DIN pin for data signal. This mode allows users to configure the down-sampling parameter *i2s_pdm_rx_clk_config_t::dn_sample_mode*. There are two down-sampling modes in PDM RX:

- *i2s_pdm_dsr_t::I2S_PDM_DSR_8S*: In this mode, the clock frequency (Fpdm) on the WS pin is `sample_rate (Fpcm) * 64`.
- *i2s_pdm_dsr_t::I2S_PDM_DSR_16S*: In this mode, the clock frequency (Fpdm) on the WS pin is `sample_rate (Fpcm) * 128`.

ADC/DAC Mode ADC and DAC modes only exist on ESP32 and are only supported on I2S0. Actually, they are two sub-modes of LCD/Camera mode. I2S0 can be routed directly to the internal analog-to-digital converter (ADC) and digital-to-analog converter (DAC). In other words, ADC and DAC peripherals can read or write continuously via I2S0 DMA. As they are not actual communication modes, the I2S driver does not implement them.

Functional Overview

The I2S driver offers the following services:

Resource Management There are three levels of resources in the I2S driver:

- `platform level`: Resources of all I2S controllers in the current target.
- `controller level`: Resources in one I2S controller.
- `channel level`: Resources of TX or RX channel in one I2S controller.

The public APIs are all channel-level APIs. The channel handle *i2s_chan_handle_t* can help users to manage the resources under a specific channel without considering the other two levels. The other two upper levels' resources are private and are managed by the driver automatically. Users can call *i2s_new_channel()* to allocate a channel handle and call *i2s_del_channel()* to delete it.

Power Management When the power management is enabled (i.e., *CONFIG_PM_ENABLE* is on), the system will adjust or stop the source clock of I2S before entering Light-sleep, thus potentially changing the I2S signals and leading to transmitting or receiving invalid data.

The I2S driver can prevent the system from changing or stopping the source clock by acquiring a power management lock. When the source clock is generated from APB, the lock type will be set to *esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX* and when the source clock is APLL (if supported), it will be set to *esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP*. Whenever the user is reading or writing via I2S (i.e., calling *i2s_channel_read()* or *i2s_channel_write()*), the driver will guarantee that the power management lock is acquired. Likewise, the driver releases the lock after the reading or writing finishes.

Finite State Machine There are three states for an I2S channel, namely, `registered`, `ready`, and `running`. Their relationship is shown in the following diagram:

The `<mode>` in the diagram can be replaced by corresponding I2S communication modes, e.g., `std` for standard two-slot mode. For more information about communication modes, please refer to the *I2S Communication Mode* section.

Data Transport The data transport of the I2S peripheral, including sending and receiving, is realized by DMA. Before transporting data, please call *i2s_channel_enable()* to enable the specific channel. When the sent or received data reaches the size of one DMA buffer, the `I2S_OUT_EOF` or `I2S_IN_SUC_EOF` interrupt will be triggered. Note that the DMA buffer size is not equal to *i2s_chan_config_t::dma_frame_num*. One frame here refers to all the sampled data in one WS circle. Therefore, `dma_buffer_size = dma_frame_num * slot_num * slot_bit_width / 8`. For the data transmitting, users can input the data by calling *i2s_channel_write()*. This function helps users to copy the data from the source buffer to the DMA TX buffer and wait for the transmission to finish. Then it will repeat until the sent bytes reach the given size. For the data receiving, the function *i2s_channel_read()* waits to receive the message queue which contains the DMA buffer address. It helps users copy the data from the DMA RX buffer to the destination buffer.

Both *i2s_channel_write()* and *i2s_channel_read()* are blocking functions. They keeps waiting until the whole source buffer is sent or the whole destination buffer is loaded, unless they exceed the max blocking time,

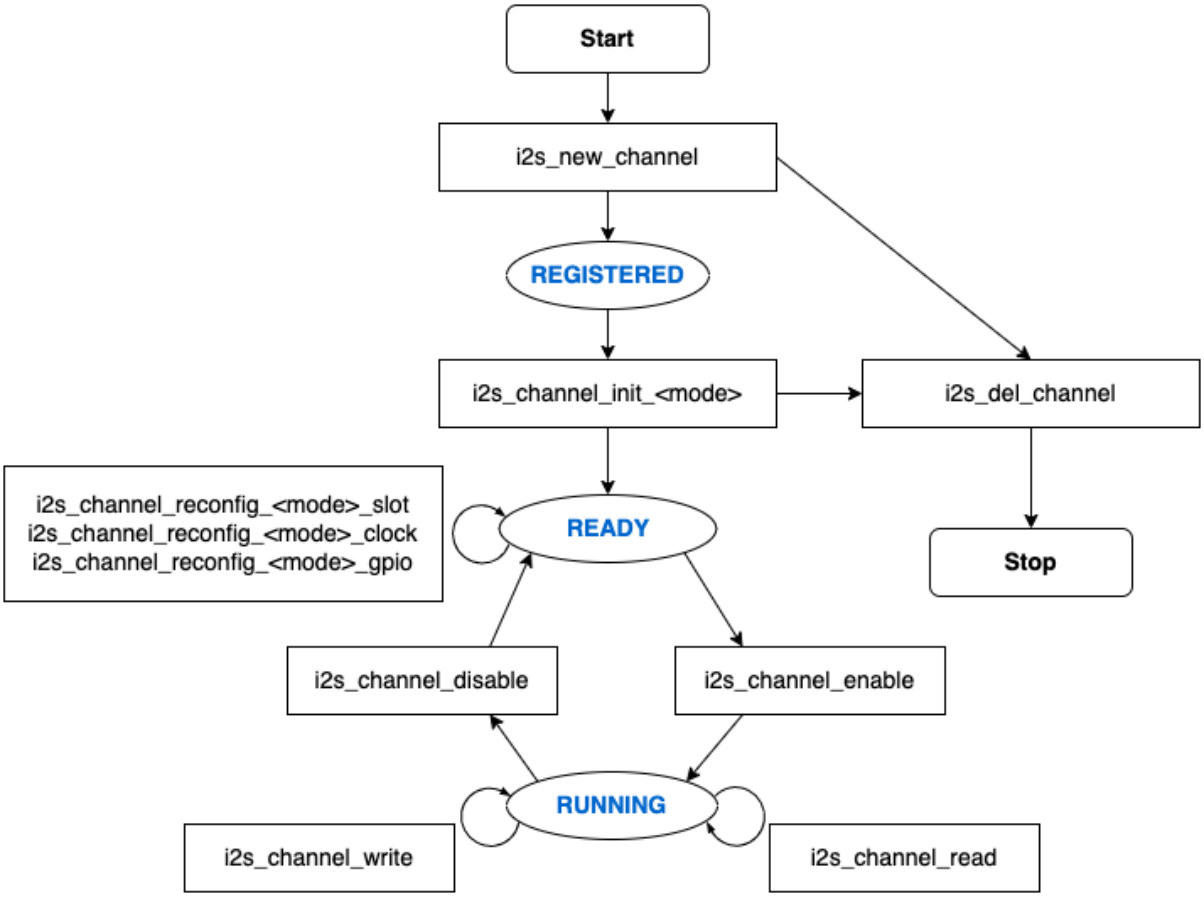


Fig. 10: I2S Finite State Machine

where the error code *ESP_ERR_TIMEOUT* returns. To send or receive data asynchronously, callbacks can be registered by `i2s_channel_register_event_callback()`. Users are able to access the DMA buffer directly in the callback function instead of transmitting or receiving by the two blocking functions. However, please be aware that it is an interrupt callback, so do not add complex logic, run floating operation, or call non-reentrant functions in the callback.

Configuration Users can initialize a channel by calling corresponding functions (i.e., `i2s_channel_init_std_mode()`, `i2s_channel_init_pdm_rx_mode()`, `i2s_channel_init_pdm_tx_mode()`, or `i2s_channel_init_tdm_mode()`) to a specific mode. If the configurations need to be updated after initialization, users have to first call `i2s_channel_disable()` to ensure that the channel has stopped, and then call corresponding 'reconfig' functions, like `i2s_channel_reconfig_std_slot()`, `i2s_channel_reconfig_std_clock()`, and `i2s_channel_reconfig_std_gpio()`.

IRAM Safe By default, the I2S interrupt will be deferred when the cache is disabled for reasons like writing/erasing flash. Thus the EOF interrupt will not get executed in time.

To avoid such case in real-time applications, you can enable the Kconfig option *CONFIG_I2S_ISR_IRAM_SAFE* that will:

1. Keep the interrupt being serviced even when the cache is disabled.
2. Place driver object into DRAM (in case it is linked to PSRAM by accident).

This will allow the interrupt to run while the cache is disabled, but will come at the cost of increased IRAM consumption.

Thread Safety All the public I2S APIs are guaranteed to be thread safe by the driver, which means users can call them from different RTOS tasks without protection by extra locks. Notice that the I2S driver uses mutex lock to ensure the thread safety, thus these APIs are not allowed to be used in ISR.

Kconfig Options

- *CONFIG_I2S_ISR_IRAM_SAFE* controls whether the default ISR handler can work when the cache is disabled. See *IRAM Safe* for more information.
- *CONFIG_I2S_SUPPRESS_DEPRECATE_WARN* controls whether to suppress the compiling warning message while using the legacy I2S driver.
- *CONFIG_I2S_ENABLE_DEBUG_LOG* is used to enable the debug log output. Enable this option will increase the firmware binary size.

Application Example

The examples of the I2S driver can be found in the directory peripherals/i2s. Here are some simple usages of each mode:

Standard TX/RX Usage Different slot communication formats can be generated by the following helper macros for standard mode. As described above, there are three formats in standard mode, and their helper macros are:

- `I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG`
- `I2S_STD_PCM_SLOT_DEFAULT_CONFIG`
- `I2S_STD_MSB_SLOT_DEFAULT_CONFIG`

The clock config helper macro is:

- `I2S_STD_CLK_DEFAULT_CONFIG`

Please refer to *Standard Mode* for information about STD API. And for more details, please refer to driver/include/driver/i2s_std.h.

STD TX Mode Take 16-bit data width for example. When the data in a `uint16_t` writing buffer are:

data 0	data 1	data 2	data 3	data 4	data 5	data 6	data 7	···
0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008	···

Here is the table of the real data on the line with different *i2s_std_slot_config_t::slot_mode* and *i2s_std_slot_config_t::slot_mask*.

data bit width	slot mode	slot mask	WS low	WS high	WS low	WS high	WS low	WS high	WS low	WS high
16 bit	mono	left	0x0002	0x0000	0x0001	0x0000	0x0004	0x0000	0x0003	0x0000
		right	0x0000	0x0002	0x0000	0x0001	0x0000	0x0004	0x0000	0x0003
		both	0x0002	0x0002	0x0001	0x0001	0x0004	0x0004	0x0003	0x0003
	stereo	left	0x0001	0x0001	0x0003	0x0003	0x0005	0x0005	0x0007	0x0007
		right	0x0002	0x0002	0x0004	0x0004	0x0006	0x0006	0x0008	0x0008
		both	0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008

Note: It is similar when the data is 32-bit width, but take care when using 8-bit and 24-bit data width. For 8-bit width, the written buffer should still use `uint16_t` (i.e., align with 2 bytes), and only the high 8 bits are valid while the low 8 bits are dropped. For 24-bit width, the buffer is supposed to use `uint32_t` (i.e., align with 4 bytes), and only the high 24 bits are valid while the low 8 bits are dropped.

Besides, for 8-bit and 16-bit mono modes, the real data on the line is swapped. To get the correct data sequence, the writing buffer needs to swap the data every two bytes.

```
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;
/* Get the default channel configuration by the helper macro.
 * This helper macro is defined in 'i2s_common.h' and shared by all the I2S␣
↪communication modes.
 * It can help to specify the I2S role and port ID */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_
↪MASTER);
/* Allocate a new TX channel and get the handle of this channel */
i2s_new_channel(&chan_cfg, &tx_handle, NULL);

/* Setting the configurations, the slot configuration and clock configuration can␣
↪be generated by the macros
 * These two helper macros are defined in 'i2s_std.h' which can only be used in␣
↪STD mode.
 * They can help to specify the slot and clock configurations for initialization␣
↪or updating */
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_
↪MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = GPIO_NUM_18,
        .din = I2S_GPIO_UNUSED,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
```

(continues on next page)

(continued from previous page)

```
        },
    },
};
/* Initialize the channel */
i2s_channel_init_std_mode(tx_handle, &std_cfg);

/* Before writing data, start the TX channel first */
i2s_channel_enable(tx_handle);
i2s_channel_write(tx_handle, src_buf, bytes_to_write, bytes_written, ticks_to_
↪wait);

/* If the configurations of slot or clock need to be updated,
 * stop the channel first and then update it */
// i2s_channel_disable(tx_handle);
// std_cfg.slot_cfg.slot_mode = I2S_SLOT_MODE_MONO; // Default is stereo
// i2s_channel_reconfig_std_slot(tx_handle, &std_cfg.slot_cfg);
// std_cfg.clk_cfg.sample_rate_hz = 96000;
// i2s_channel_reconfig_std_clock(tx_handle, &std_cfg.clk_cfg);

/* Have to stop the channel before deleting it */
i2s_channel_disable(tx_handle);
/* If the handle is not needed any more, delete it to release the channel␣
↪resources */
i2s_del_channel(tx_handle);
```

STD RX Mode Taking 16-bit data width for example, when the data on the line are:

WS low	WS high	WS low	WS high	WS low	WS high	WS low	WS high	···
0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008	···

Here is the table of the data received in the buffer with different *i2s_std_slot_config_t::slot_mode* and *i2s_std_slot_config_t::slot_mask*.

data bit width	slot mode	slot mask	data 0	data 1	data 2	data 3	data 4	data 5	data 6	data 7
16 bit	mono	left	0x0001	0x0000	0x0005	0x0003	0x0009	0x0007	0x000d	0x000b
		right	0x0002	0x0000	0x0006	0x0004	0x000a	0x0008	0x000e	0x000c
	stereo	any	0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008

Note: The receive case is a little bit complicated on ESP32. Firstly, when the data width is 8-bit or 24-bit, the received data will still align with two bytes or four bytes, which means that the valid data are put in the high 8 bits in every two bytes and high 24 bits in every four bytes. For example, the received data will be `0x5A00` when the data on the line is `0x5A` in 8-bit width, and `0x0000 5A00` if the data on the line is `0x00 005A`. Secondly, for the 8-bit or 16-bit mono case, the data in buffer is swapped every two data, so it may be necessary to manually swap the data back to the correct order.

```
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t rx_handle;
/* Get the default channel configuration by helper macro.
 * This helper macro is defined in 'i2s_common.h' and shared by all the I2S␣
↪communication modes.
 * It can help to specify the I2S role and port ID */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_
↪MASTER);
```

(continues on next page)

(continued from previous page)

```
/* Allocate a new RX channel and get the handle of this channel */
i2s_new_channel(&chan_cfg, NULL, &rx_handle);

/* Setting the configurations, the slot configuration and clock configuration can be generated by the macros
 * These two helper macros are defined in 'i2s_std.h' which can only be used in STD mode.
 * They can help to specify the slot and clock configurations for initialization or updating */
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = I2S_GPIO_UNUSED,
        .din = GPIO_NUM_19,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
/* Initialize the channel */
i2s_channel_init_std_mode(rx_handle, &std_cfg);

/* Before reading data, start the RX channel first */
i2s_channel_enable(rx_handle);
i2s_channel_read(rx_handle, desc_buf, bytes_to_read, bytes_read, ticks_to_wait);

/* Have to stop the channel before deleting it */
i2s_channel_disable(rx_handle);
/* If the handle is not needed any more, delete it to release the channel resources */
i2s_del_channel(rx_handle);
```

PDM TX Usage For PDM mode in TX channel, the slot configuration helper macro is:

- *I2S_PDM_TX_SLOT_DEFAULT_CONFIG*

The clock configuration helper macro is:

- *I2S_PDM_TX_CLK_DEFAULT_CONFIG*

Please refer to *PDM Mode* for information about PDM TX API. And for more details, please refer to driver/include/driver/i2s_pdm.h.

The PDM data width is fixed to 16-bit. When the data in an `int16_t` writing buffer is:

data 0	data 1	data 2	data 3	data 4	data 5	data 6	data 7	···
0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008	···

Here is the table of the real data on the line with different *i2s_pdm_tx_slot_config_t::slot_mode* and *i2s_pdm_tx_slot_config_t::slot_mask* (The PDM format on the line is transferred to PCM format for better comprehension).

slot mode	slot mask	left	right	left	right	left	right	left	right
mono	left	0x0001	0x0000	0x0002	0x0000	0x0003	0x0000	0x0004	0x0000
	right	0x0000	0x0001	0x0000	0x0002	0x0000	0x0003	0x0000	0x0004
	both	0x0001	0x0001	0x0002	0x0002	0x0003	0x0003	0x0004	0x0004
stereo	left	0x0001	0x0001	0x0003	0x0003	0x0005	0x0005	0x0007	0x0007
	right	0x0002	0x0002	0x0004	0x0004	0x0006	0x0006	0x0008	0x0008
	both	0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008

```
#include "driver/i2s_pdm.h"
#include "driver/gpio.h"

/* Allocate an I2S TX channel */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_0, I2S_ROLE_
↪MASTER);
i2s_new_channel(&chan_cfg, &tx_handle, NULL);

/* Init the channel into PDM TX mode */
i2s_pdm_tx_config_t pdm_tx_cfg = {
    .clk_cfg = I2S_PDM_TX_CLK_DEFAULT_CONFIG(36000),
    .slot_cfg = I2S_PDM_TX_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_
↪MODE_MONO),
    .gpio_cfg = {
        .clk = GPIO_NUM_5,
        .dout = GPIO_NUM_18,
        .invert_flags = {
            .clk_inv = false,
        },
    },
};
i2s_channel_init_pdm_tx_mode(tx_handle, &pdm_tx_cfg);

...
```

PDM RX usage For PDM mode in RX channel, the slot configuration helper macro is:

- *I2S_PDM_RX_SLOT_DEFAULT_CONFIG*

The clock configuration helper macro is:

- *I2S_PDM_RX_CLK_DEFAULT_CONFIG*

Please refer to *PDM Mode* for information about PDM RX API. And for more details, please refer to driver/include/driver/i2s_pdm.h.

The PDM data width is fixed to 16-bit. When the data on the line (The PDM format on the line is transferred to PCM format for easier comprehension) is:

left	right	left	right	left	right	left	right	···
0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008	···

Here is the table of the data received in a 'int16_t' buffer with different *i2s_pdm_rx_slot_config_t::slot_mode* and *i2s_pdm_rx_slot_config_t::slot_mask*.

slot mode	slot mask	data 0	data 1	data 2	data 3	data 4	data 5	data 6	data 7
mono	left	0x0001	0x0003	0x0005	0x0007	0x0009	0x000b	0x000d	0x000f
	right	0x0002	0x0004	0x0006	0x0008	0x000a	0x000c	0x000e	0x0010
stereo	both	0x0001	0x0002	0x0003	0x0004	0x0005	0x0006	0x0007	0x0008

```
#include "driver/i2s_pdm.h"
#include "driver/gpio.h"

i2s_chan_handle_t rx_handle;

/* Allocate an I2S RX channel */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_0, I2S_ROLE_
↪MASTER);
i2s_new_channel(&chan_cfg, NULL, &rx_handle);

/* Init the channel into PDM RX mode */
i2s_pdm_rx_config_t pdm_rx_cfg = {
    .clk_cfg = I2S_PDM_RX_CLK_DEFAULT_CONFIG(36000),
    .slot_cfg = I2S_PDM_RX_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_
↪MODE_MONO),
    .gpio_cfg = {
        .clk = GPIO_NUM_5,
        .din = GPIO_NUM_19,
        .invert_flags = {
            .clk_inv = false,
        },
    },
};
i2s_channel_init_pdm_rx_mode(rx_handle, &pdm_rx_cfg);

...
```

Full-duplex Full-duplex mode registers TX and RX channel in an I2S port at the same time, and the channels share the BCLK and WS signals. Currently, STD and TDM communication modes supports full-duplex mode in the following way, but PDM full-duplex is not supported because due to different PDM TX and RX clocks.

Note that one handle can only stand for one channel. Therefore, it is still necessary to configure the slot and clock for both TX and RX channels one by one.

Here is an example of how to allocate a pair of full-duplex channels:

```
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;
i2s_chan_handle_t rx_handle;

/* Allocate a pair of I2S channel */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_
↪MASTER);
/* Allocate for TX and RX channel at the same time, then they will work in full-
↪duplex mode */
i2s_new_channel(&chan_cfg, &tx_handle, &rx_handle);

/* Set the configurations for BOTH TWO channels, since TX and RX channel have to␣
↪be same in full-duplex mode */
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(32000),
    .slot_cfg = I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_
↪SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = GPIO_NUM_18,
        .din = GPIO_NUM_19,
```

(continues on next page)

(continued from previous page)

```
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
i2s_channel_init_std_mode(tx_handle, &std_cfg);
i2s_channel_init_std_mode(rx_handle, &std_cfg);

i2s_channel_enable(tx_handle);
i2s_channel_enable(rx_handle);

...
```

Simplex Mode To allocate a channel handle in simplex mode, *i2s_new_channel()* should be called for each channel. The clock and GPIO pins of TX/RX channel on ESP32 are not independent, so the TX and RX channel cannot coexist on the same I2S port in simplex mode.

```
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;
i2s_chan_handle_t rx_handle;

i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_
↪MASTER);
i2s_new_channel(&chan_cfg, &tx_handle, NULL);
i2s_std_config_t std_tx_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),
    .slot_cfg = I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_
↪SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = GPIO_NUM_0,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = GPIO_NUM_18,
        .din = I2S_GPIO_UNUSED,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
/* Initialize the channel */
i2s_channel_init_std_mode(tx_handle, &std_tx_cfg);
i2s_channel_enable(tx_handle);

/* RX channel will be registered on another I2S, if no other available I2S unit␣
↪found
 * it will return ESP_ERR_NOT_FOUND */
i2s_new_channel(&chan_cfg, NULL, &rx_handle);
i2s_std_config_t std_rx_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(16000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_
↪MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_6,
```

(continues on next page)

(continued from previous page)

```
        .ws = GPIO_NUM_7,
        .dout = I2S_GPIO_UNUSED,
        .din = GPIO_NUM_19,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
i2s_channel_init_std_mode(rx_handle, &std_rx_cfg);
i2s_channel_enable(rx_handle);
```

Application Notes

How to Prevent Data Lost For applications that need a high frequency sample rate, the massive data throughput may cause data lost. Users can receive data lost event by registering the ISR callback function to receive the event queue:

```
static IRAM_ATTR bool i2s_rx_queue_overflow_callback(i2s_chan_handle_t␣
↪handle, i2s_event_data_t *event, void *user_ctx)
{
    // handle RX queue overflow event ...
    return false;
}

i2s_event_callbacks_t cbs = {
    .on_recv = NULL,
    .on_recv_q_ovf = i2s_rx_queue_overflow_callback,
    .on_sent = NULL,
    .on_send_q_ovf = NULL,
};
TEST_ESP_OK(i2s_channel_register_event_callback(rx_handle, &cbs, NULL));
```

Please follow these steps to prevent data lost:

1. Determine the interrupt interval. Generally, when data lost happens, the bigger the interval, the better, which helps to reduce the interrupt times. This means `dma_frame_num` should be as big as possible while the DMA buffer size is below the maximum value of 4092. The relationships are:

   ```
   interrupt_interval(unit: sec) = dma_frame_num / sample_rate
   dma_buffer_size = dma_frame_num * slot_num * data_bit_width / 8 <= 4092
   ```

2. Determine `dma_desc_num`. `dma_desc_num` is decided by the maximum time of `i2s_channel_read` polling cycle. All the received data is supposed to be stored between two `i2s_channel_read`. This cycle can be measured by a timer or an outputting GPIO signal. The relationship is:

   ```
   dma_desc_num > polling_cycle / interrupt_interval
   ```

3. Determine the receiving buffer size. The receiving buffer offered by users in `i2s_channel_read` should be able to take all the data in all DMA buffers, which means that it should be larger than the total size of all the DMA buffers:

   ```
   recv_buffer_size > dma_desc_num * dma_buffer_size
   ```

For example, if there is an I2S application, and the known values are:

```
sample_rate = 144000 Hz
data_bit_width = 32 bits
slot_num = 2
polling_cycle = 10 ms
```

Then the parameters `dma_frame_num`, `dma_desc_num`, and `recv_buf_size` can be calculated as follows:

```
dma_frame_num * slot_num * data_bit_width / 8 = dma_buffer_size <= 4092
dma_frame_num <= 511
interrupt_interval = dma_frame_num / sample_rate = 511 / 144000 = 0.003549 s = 3.
↪549 ms
dma_desc_num > polling_cycle / interrupt_interval = cell(10 / 3.549) = cell(2.818)␣
↪= 3
recv_buffer_size > dma_desc_num * dma_buffer_size = 3 * 4092 = 12276 bytes
```

API Reference

Standard Mode

Header File

- components/driver/include/driver/i2s_std.h

Functions

esp_err_t **i2s_channel_init_std_mode**(*i2s_chan_handle_t* handle, const *i2s_std_config_t* *std_cfg)

Initialize i2s channel to standard mode.

Note: Only allowed to be called when the channel state is REGISTERED, (i.e., channel has been allocated, but not initialized) and the state will be updated to READY if initialization success, otherwise the state will return to REGISTERED.

Parameters

- **handle** –**[in]** I2S channel handler
- **std_cfg** –**[in]** Configurations for standard mode, including clock, slot and gpio The clock configuration can be generated by the helper macro `I2S_STD_CLK_DEFAULT_CONFIG` The slot configuration can be generated by the helper macro `I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG`, `I2S_STD_PCM_SLOT_DEFAULT_CONFIG` or `I2S_STD_MSB_SLOT_DEFAULT_CONFIG`

Returns

- ESP_OK Initialize successfully
- ESP_ERR_NO_MEM No memory for storing the channel information
- ESP_ERR_INVALID_ARG NULL pointer or invalid configuration
- ESP_ERR_INVALID_STATE This channel is not registered

esp_err_t **i2s_channel_reconfig_std_clock**(*i2s_chan_handle_t* handle, const *i2s_std_clk_config_t* *clk_cfg)

Reconfigure the I2S clock for standard mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to standard mode, i.e., 'i2s_channel_init_std_mode' has been called before reconfiguring

---

Parameters

- **handle** –**[in]** I2S channel handler

- **clk_cfg** –**[in]** Standard mode clock configuration, can be generated by I2S_STD_CLK_DEFAULT_CONFIG

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not standard mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_std_slot**(*i2s_chan_handle_t* handle, const *i2s_std_slot_config_t* *slot_cfg)

Reconfigure the I2S slot for standard mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to standard mode, i.e., 'i2s_channel_init_std_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S channel handler
- **slot_cfg** –**[in]** Standard mode slot configuration, can be generated by I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG, I2S_STD_PCM_SLOT_DEFAULT_CONFIG and I2S_STD_MSB_SLOT_DEFAULT_CONFIG.

Returns

- ESP_OK Set clock successfully
- ESP_ERR_NO_MEM No memory for DMA buffer
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not standard mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_std_gpio**(*i2s_chan_handle_t* handle, const *i2s_std_gpio_config_t* *gpio_cfg)

Reconfigure the I2S gpio for standard mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to standard mode, i.e., 'i2s_channel_init_std_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S channel handler
- **gpio_cfg** –**[in]** Standard mode gpio configuration, specified by user

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not standard mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

Structures

struct **i2s_std_slot_config_t**

I2S slot configuration for standard mode.

Public Members

i2s_data_bit_width_t **data_bit_width**

I2S sample data bit width (valid data bits per sample)

i2s_slot_bit_width_t **slot_bit_width**

I2S slot bit width (total bits per slot)

i2s_slot_mode_t **slot_mode**

Set mono or stereo mode with I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO In TX direction, mono means the written buffer contains only one slot data and stereo means the written buffer contains both left and right data

i2s_std_slot_mask_t **slot_mask**

Select the left, right or both slot

uint32_t **ws_width**

WS signal width (i.e. the number of bclk ticks that ws signal is high)

bool **ws_pol**

WS signal polarity, set true to enable high lever first

bool **bit_shift**

Set to enable bit shift in Philips mode

bool **msb_right**

Set to place right channel data at the MSB in the FIFO

struct **i2s_std_clk_config_t**

I2S clock configuration for standard mode.

Public Members

uint32_t **sample_rate_hz**

I2S sample rate

i2s_clock_src_t **clk_src**

Choose clock source

i2s_mclk_multiple_t **mclk_multiple**

The multiple of mclk to the sample rate Default is 256 in the helper macro, it can satisfy most of cases, but please set this field a multiple of '3' (like 384) when using 24-bit data width, otherwise the sample rate might be inaccurate

struct **i2s_std_gpio_config_t**

I2S standard mode GPIO pins configuration.

Public Members

gpio_num_t **mclk**

MCK pin, output

gpio_num_t **bclk**

BCK pin, input in slave role, output in master role

gpio_num_t **ws**

WS pin, input in slave role, output in master role

gpio_num_t **dout**

DATA pin, output

gpio_num_t **din**

DATA pin, input

uint32_t **mclk_inv**

Set 1 to invert the mclk output

uint32_t **bclk_inv**

Set 1 to invert the bclk input/output

uint32_t **ws_inv**

Set 1 to invert the ws input/output

struct *i2s_std_gpio_config_t*::[anonymous] **invert_flags**

GPIO pin invert flags

struct **i2s_std_config_t**

I2S standard mode major configuration that including clock/slot/gpio configuration.

Public Members

i2s_std_clk_config_t **clk_cfg**

Standard mode clock configuration, can be generated by macro I2S_STD_CLK_DEFAULT_CONFIG

i2s_std_slot_config_t **slot_cfg**

Standard mode slot configuration, can be generated by macros I2S_STD_[mode]_SLOT_DEFAULT_CONFIG, [mode] can be replaced with PHILIPS/MSB/PCM

i2s_std_gpio_config_t **gpio_cfg**

Standard mode gpio configuration, specified by user

Macros

I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG(bits_per_sample, mono_or_stereo)

Philips format in 2 slots.

This file is specified for I2S standard communication mode Features:

- Philips/MSB/PCM are supported in standard mode
- Fixed to 2 slots

Parameters
- **bits_per_sample** –i2s data bit width
- **mono_or_stereo** –I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

I2S_STD_PCM_SLOT_DEFAULT_CONFIG(bits_per_sample, mono_or_stereo)

PCM(short) format in 2 slots.

Note: PCM(long) is same as philips in 2 slots

Parameters
- **bits_per_sample** –i2s data bit width
- **mono_or_stereo** –I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

I2S_STD_MSB_SLOT_DEFAULT_CONFIG(bits_per_sample, mono_or_stereo)

MSB format in 2 slots.

Parameters
- **bits_per_sample** –i2s data bit width
- **mono_or_stereo** –I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

I2S_STD_CLK_DEFAULT_CONFIG(rate)

i2s default standard clock configuration

Note: Please set the mclk_multiple to I2S_MCLK_MULTIPLE_384 while using 24 bits data width Otherwise the sample rate might be imprecise since the bclk division is not a integer

Parameters
- **rate** –sample rate

PDM Mode

Header File

- components/driver/include/driver/i2s_pdm.h

Functions

esp_err_t **i2s_channel_init_pdm_rx_mode**(*i2s_chan_handle_t* handle, const *i2s_pdm_rx_config_t* *pdm_rx_cfg)

Initialize i2s channel to PDM RX mode.

Note: Only allowed to be called when the channel state is REGISTERED, (i.e., channel has been allocated, but not initialized) and the state will be updated to READY if initialization success, otherwise the state will return to REGISTERED.

Parameters
- **handle** –**[in]** I2S rx channel handler
- **pdm_rx_cfg** –**[in]** Configurations for PDM RX mode, including clock, slot and gpio The clock configuration can be generated by the helper macro `I2S_PDM_RX_CLK_DEFAULT_CONFIG` The slot configuration can be generated by the helper macro `I2S_PDM_RX_SLOT_DEFAULT_CONFIG`

Returns

- ESP_OK Initialize successfully
- ESP_ERR_NO_MEM No memory for storing the channel information
- ESP_ERR_INVALID_ARG NULL pointer or invalid configuration
- ESP_ERR_INVALID_STATE This channel is not registered

esp_err_t **i2s_channel_reconfig_pdm_rx_clock**(*i2s_chan_handle_t* handle, const *i2s_pdm_rx_clk_config_t* *clk_cfg)

Reconfigure the I2S clock for PDM RX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM RX mode, i.e., 'i2s_channel_init_pdm_rx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S rx channel handler
- **clk_cfg** –**[in]** PDM RX mode clock configuration, can be generated by `I2S_PDM_RX_CLK_DEFAULT_CONFIG`

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_pdm_rx_slot**(*i2s_chan_handle_t* handle, const *i2s_pdm_rx_slot_config_t* *slot_cfg)

Reconfigure the I2S slot for PDM RX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM RX mode, i.e., 'i2s_channel_init_pdm_rx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S rx channel handler
- **slot_cfg** –**[in]** PDM RX mode slot configuration, can be generated by `I2S_PDM_RX_SLOT_DEFAULT_CONFIG`

Returns

- ESP_OK Set clock successfully
- ESP_ERR_NO_MEM No memory for DMA buffer
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_pdm_rx_gpio**(*i2s_chan_handle_t* handle, const *i2s_pdm_rx_gpio_config_t* *gpio_cfg)

Reconfigure the I2S gpio for PDM RX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM RX mode, i.e., 'i2s_channel_init_pdm_rx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S rx channel handler
- **gpio_cfg** –**[in]** PDM RX mode gpio configuration, specified by user

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_init_pdm_tx_mode**(*i2s_chan_handle_t* handle, const *i2s_pdm_tx_config_t* *pdm_tx_cfg)

Initialize i2s channel to PDM TX mode.

Note: Only allowed to be called when the channel state is REGISTERED, (i.e., channel has been allocated, but not initialized) and the state will be updated to READY if initialization success, otherwise the state will return to REGISTERED.

Parameters

- **handle** –**[in]** I2S tx channel handler
- **pdm_tx_cfg** –**[in]** Configurations for PDM TX mode, including clock, slot and gpio The clock configuration can be generated by the helper macro `I2S_PDM_TX_CLK_DEFAULT_CONFIG` The slot configuration can be generated by the helper macro `I2S_PDM_TX_SLOT_DEFAULT_CONFIG`

Returns

- ESP_OK Initialize successfully
- ESP_ERR_NO_MEM No memory for storing the channel information
- ESP_ERR_INVALID_ARG NULL pointer or invalid configuration
- ESP_ERR_INVALID_STATE This channel is not registered

esp_err_t **i2s_channel_reconfig_pdm_tx_clock**(*i2s_chan_handle_t* handle, const *i2s_pdm_tx_clk_config_t* *clk_cfg)

Reconfigure the I2S clock for PDM TX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM TX mode, i.e., 'i2s_channel_init_pdm_tx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S tx channel handler
- **clk_cfg** –**[in]** PDM TX mode clock configuration, can be generated by `I2S_PDM_TX_CLK_DEFAULT_CONFIG`

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_pdm_tx_slot**(*i2s_chan_handle_t* handle, const *i2s_pdm_tx_slot_config_t* *slot_cfg)

Reconfigure the I2S slot for PDM TX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM TX mode, i.e., 'i2s_channel_init_pdm_tx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S tx channel handler
- **slot_cfg** –**[in]** PDM TX mode slot configuration, can be generated by `I2S_PDM_TX_SLOT_DEFAULT_CONFIG`

Returns

- ESP_OK Set clock successfully
- ESP_ERR_NO_MEM No memory for DMA buffer
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

esp_err_t **i2s_channel_reconfig_pdm_tx_gpio**(*i2s_chan_handle_t* handle, const *i2s_pdm_tx_gpio_config_t* *gpio_cfg)

Reconfigure the I2S gpio for PDM TX mode.

Note: Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won't change the state. 'i2s_channel_disable' should be called before calling this function if i2s has started.

Note: The input channel handle has to be initialized to PDM TX mode, i.e., 'i2s_channel_init_pdm_tx_mode' has been called before reconfiguring

Parameters

- **handle** –**[in]** I2S tx channel handler
- **gpio_cfg** –**[in]** PDM TX mode gpio configuration, specified by user

Returns

- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not PDM mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

Structures

struct **i2s_pdm_rx_slot_config_t**

I2S slot configuration for pdm rx mode.

Public Members

i2s_data_bit_width_t **data_bit_width**

I2S sample data bit width (valid data bits per sample), only support 16 bits for PDM mode

i2s_slot_bit_width_t **slot_bit_width**

I2S slot bit width (total bits per slot) , only support 16 bits for PDM mode

i2s_slot_mode_t **slot_mode**

Set mono or stereo mode with I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

i2s_pdm_slot_mask_t **slot_mask**

Choose the slots to activate

struct **i2s_pdm_rx_clk_config_t**

I2S clock configuration for pdm rx mode.

Public Members

uint32_t **sample_rate_hz**

I2S sample rate

i2s_clock_src_t **clk_src**

Choose clock source

i2s_mclk_multiple_t **mclk_multiple**

The multiple of mclk to the sample rate

i2s_pdm_dsr_t **dn_sample_mode**

Down-sampling rate mode

struct **i2s_pdm_rx_gpio_config_t**

I2S PDM tx mode GPIO pins configuration.

Public Members

gpio_num_t **clk**

PDM clk pin, output

gpio_num_t **din**

DATA pin, input

uint32_t **clk_inv**

Set 1 to invert the clk output

struct *i2s_pdm_rx_gpio_config_t*::[anonymous] **invert_flags**

GPIO pin invert flags

struct **i2s_pdm_rx_config_t**

I2S PDM RX mode major configuration that including clock/slot/gpio configuration.

Public Members

i2s_pdm_rx_clk_config_t **clk_cfg**

PDM RX clock configurations, can be generated by macro I2S_PDM_RX_CLK_DEFAULT_CONFIG

i2s_pdm_rx_slot_config_t **slot_cfg**

PDM RX slot configurations, can be generated by macro I2S_PDM_RX_SLOT_DEFAULT_CONFIG

i2s_pdm_rx_gpio_config_t **gpio_cfg**

PDM RX slot configurations, specified by user

struct **i2s_pdm_tx_slot_config_t**

I2S slot configuration for pdm tx mode.

Public Members

i2s_data_bit_width_t **data_bit_width**

I2S sample data bit width (valid data bits per sample), only support 16 bits for PDM mode

i2s_slot_bit_width_t **slot_bit_width**

I2S slot bit width (total bits per slot), only support 16 bits for PDM mode

i2s_slot_mode_t **slot_mode**

Set mono or stereo mode with I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO For PDM TX mode, mono means the data buffer only contains one slot data, Stereo means the data buffer contains two slots data

i2s_pdm_slot_mask_t **slot_mask**

Slot mask to choose left or right slot

uint32_t **sd_prescale**

Sigma-delta filter prescale

i2s_pdm_sig_scale_t **sd_scale**

Sigma-delta filter scaling value

i2s_pdm_sig_scale_t **hp_scale**

High pass filter scaling value

i2s_pdm_sig_scale_t **lp_scale**

Low pass filter scaling value

i2s_pdm_sig_scale_t **sinc_scale**

Sinc filter scaling value

struct **i2s_pdm_tx_clk_config_t**

I2S clock configuration for pdm tx mode.

Public Members

uint32_t **sample_rate_hz**

I2S sample rate, not suggest to exceed 48000 Hz, otherwise more glitches and noise may appear

i2s_clock_src_t **clk_src**

Choose clock source

i2s_mclk_multiple_t **mclk_multiple**

The multiple of mclk to the sample rate

uint32_t **up_sample_fp**

Up-sampling param fp

uint32_t **up_sample_fs**

Up-sampling param fs, not allowed to be greater than 480

struct **i2s_pdm_tx_gpio_config_t**

I2S PDM tx mode GPIO pins configuration.

Public Members

gpio_num_t **clk**

PDM clk pin, output

gpio_num_t **dout**

DATA pin, output

uint32_t **clk_inv**

Set 1 to invert the clk output

struct *i2s_pdm_tx_gpio_config_t*::[anonymous] **invert_flags**

GPIO pin invert flags

struct **i2s_pdm_tx_config_t**

I2S PDM TX mode major configuration that including clock/slot/gpio configuration.

Public Members

i2s_pdm_tx_clk_config_t **clk_cfg**

PDM TX clock configurations, can be generated by macro I2S_PDM_TX_CLK_DEFAULT_CONFIG

i2s_pdm_tx_slot_config_t **slot_cfg**

PDM TX slot configurations, can be generated by macro I2S_PDM_TX_SLOT_DEFAULT_CONFIG

i2s_pdm_tx_gpio_config_t **gpio_cfg**

PDM TX gpio configurations, specified by user

Macros

I2S_PDM_RX_SLOT_DEFAULT_CONFIG(bits_per_sample, mono_or_stereo)

PDM format in 2 slots(RX)

This file is specified for I2S PDM communication mode Features:

- Only support PDM tx/rx mode
- Fixed to 2 slots
- Data bit width only support 16 bits

Parameters

- **bits_per_sample** –i2s data bit width, only support 16 bits for PDM mode
- **mono_or_stereo** –I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

I2S_PDM_RX_CLK_DEFAULT_CONFIG(rate)

i2s default pdm rx clock configuration

Parameters

- **rate** –sample rate

I2S_PDM_TX_SLOT_DEFAULT_CONFIG(bits_per_sample, mono_or_stereo)

PDM style in 2 slots(TX)

Parameters

- **bits_per_sample** –i2s data bit width, only support 16 bits for PDM mode
- **mono_or_stereo** –I2S_SLOT_MODE_MONO or I2S_SLOT_MODE_STEREO

I2S_PDM_TX_CLK_DEFAULT_CONFIG(rate)

i2s default pdm tx clock configuration

Note: TX PDM can only be set to the following two up-sampling rate configurations: 1: fp = 960, fs = sample_rate_hz / 100, in this case, Fpdm = 128*48000 2: fp = 960, fs = 480, in this case, Fpdm = 128*Fpcm = 128*sample_rate_hz If the pdm receiver do not care the pdm serial clock, it's recommended set Fpdm = 128*48000. Otherwise, the second configuration should be adopted.

Parameters

- **rate** –sample rate (not suggest to exceed 48000 Hz, otherwise more glitches and noise may appear)

I2S Driver

Header File

- components/driver/include/driver/i2s_common.h

Functions

esp_err_t **i2s_new_channel**(const *i2s_chan_config_t* *chan_cfg, *i2s_chan_handle_t* *ret_tx_handle, *i2s_chan_handle_t* *ret_rx_handle)

Allocate new I2S channel(s)

Note: The new created I2S channel handle will be REGISTERED state after it is allocated successfully.

Note: When the port id in channel configuration is I2S_NUM_AUTO, driver will allocate I2S port automatically on one of the i2s controller, otherwise driver will try to allocate the new channel on the selected port.

Note: If both tx_handle and rx_handle are not NULL, it means this I2S controller will work at full-duplex mode, the rx and tx channels will be allocated on a same I2S port in this case. Note that some configurations of tx/rx channel are shared on ESP32 and ESP32S2, so please make sure they are working at same condition and under same status(start/stop). Currently, full-duplex mode can't guarantee tx/rx channels write/read synchronously, they can only share the clock signals for now.

Note: If tx_handle OR rx_handle is NULL, it means this I2S controller will work at simplex mode. For ESP32 and ESP32S2, the whole I2S controller (i.e. both rx and tx channel) will be occupied, even if only one of rx or tx channel is registered. For the other targets, another channel on this controller will still available.

Parameters

- **chan_cfg** –**[in]** I2S controller channel configurations
- **ret_tx_handle** –**[out]** I2S channel handler used for managing the sending channel(optional)
- **ret_rx_handle** –**[out]** I2S channel handler used for managing the receiving channel(optional)

Returns

- ESP_OK Allocate new channel(s) success
- ESP_ERR_NOT_SUPPORTED The communication mode is not supported on the current chip
- ESP_ERR_INVALID_ARG NULL pointer or illegal parameter in *i2s_chan_config_t*
- ESP_ERR_NOT_FOUND No available I2S channel found

esp_err_t **i2s_del_channel**(*i2s_chan_handle_t* handle)

Delete the i2s channel.

Note: Only allowed to be called when the i2s channel is at REGISTERED or READY state (i.e., it should stop before deleting it).

Note: Resource will be free automatically if all channels in one port are deleted

---

Parameters **handle** –**[in]** I2S channel handler

- ESP_OK Delete successfully
- ESP_ERR_INVALID_ARG NULL pointer

esp_err_t **i2s_channel_get_info**(*i2s_chan_handle_t* handle, *i2s_chan_info_t* *chan_info)

Get I2S channel information.

Parameters

- **handle** –**[in]** I2S channel handler
- **chan_info** –**[out]** I2S channel basic information

Returns

- ESP_OK Get i2s channel information success
- ESP_ERR_NOT_FOUND The input handle doesn't match any registered I2S channels, it may not an i2s channel handle or not available any more
- ESP_ERR_INVALID_ARG The input handle or chan_info pointer is NULL

esp_err_t **i2s_channel_enable**(*i2s_chan_handle_t* handle)

Enable the i2s channel.

Note: Only allowed to be called when the channel state is READY, (i.e., channel has been initialized, but not started) the channel will enter RUNNING state once it is enabled successfully.

Note: Enable the channel can start the I2S communication on hardware. It will start outputting bclk and ws signal. For mclk signal, it will start to output when initialization is finished

---

Parameters **handle** –**[in]** I2S channel handler

- ESP_OK Start successfully
- ESP_ERR_INVALID_ARG NULL pointer
- ESP_ERR_INVALID_STATE This channel has not initialized or already started

esp_err_t **i2s_channel_disable**(*i2s_chan_handle_t* handle)

Disable the i2s channel.

Note: Only allowed to be called when the channel state is READY / RUNNING, (i.e., channel has been initialized) the channel will enter READY state once it is disabled successfully.

Note: Disable the channel can stop the I2S communication on hardware. It will stop bclk and ws signal but not mclk signal

---

Parameters **handle** –**[in]** I2S channel handler

Returns

- ESP_OK Stop successfully
- ESP_ERR_INVALID_ARG NULL pointer
- ESP_ERR_INVALID_STATE This channel has not stated

esp_err_t **i2s_channel_write**(*i2s_chan_handle_t* handle, const void *src, size_t size, size_t *bytes_written, uint32_t timeout_ms)

I2S write data.

Note: Only allowed to be called when the channel state is RUNNING, (i.e., tx channel has been started and is not writing now) but the RUNNING only stands for the software state, it doesn't mean there is no the signal transporting on line.

Parameters

- **handle** –**[in]** I2S channel handler
- **src** –**[in]** The pointer of sent data buffer
- **size** –**[in]** Max data buffer length
- **bytes_written** –**[out]** Byte number that actually be sent

- **timeout_ms** –**[in]** Max block time

Returns

- ESP_OK Write successfully
- ESP_ERR_INVALID_ARG NULL pointer or this handle is not tx handle
- ESP_ERR_TIMEOUT Writing timeout, no writing event received from ISR within ticks_to_wait
- ESP_ERR_INVALID_STATE I2S is not ready to write

esp_err_t **i2s_channel_read**(*i2s_chan_handle_t* handle, void *dest, size_t size, size_t *bytes_read, uint32_t timeout_ms)

I2S read data.

Note: Only allowed to be called when the channel state is RUNNING but the RUNNING only stands for the software state, it doesn't mean there is no the signal transporting on line.

Parameters

- **handle** –**[in]** I2S channel handler
- **dest** –**[in]** The pointer of receiving data buffer
- **size** –**[in]** Max data buffer length
- **bytes_read** –**[out]** Byte number that actually be read
- **timeout_ms** –**[in]** Max block time

Returns

- ESP_OK Read successfully
- ESP_ERR_INVALID_ARG NULL pointer or this handle is not rx handle
- ESP_ERR_TIMEOUT Reading timeout, no reading event received from ISR within ticks_to_wait
- ESP_ERR_INVALID_STATE I2S is not ready to read

esp_err_t **i2s_channel_register_event_callback**(*i2s_chan_handle_t* handle, const *i2s_event_callbacks_t* *callbacks, void *user_data)

Set event callbacks for I2S channel.

Note: Only allowed to be called when the channel state is REGISTARED / READY, (i.e., before channel starts)

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `callbacks` structure to NULL.

Note: When CONFIG_I2S_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM or internal RAM as well.

Parameters

- **handle** –**[in]** I2S channel handler
- **callbacks** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK Set event callbacks successfully
- ESP_ERR_INVALID_ARG Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE Set event callbacks failed because the current channel state is not REGISTARED or READY

Structures

struct **i2s_event_callbacks_t**

Group of I2S callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_I2S_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

i2s_isr_callback_t **on_recv**

Callback of data received event, only for rx channel The event data includes DMA buffer address and size that just finished receiving data

i2s_isr_callback_t **on_recv_q_ovf**

Callback of receiving queue overflowed event, only for rx channel The event data includes buffer size that has been overwritten

i2s_isr_callback_t **on_sent**

Callback of data sent event, only for tx channel The event data includes DMA buffer address and size that just finished sending data

i2s_isr_callback_t **on_send_q_ovf**

Callback of sending queue overflowed event, only for tx channel The event data includes buffer size that has been overwritten

struct **i2s_chan_config_t**

I2S controller channel configuration.

Public Members

i2s_port_t **id**

I2S port id

i2s_role_t **role**

I2S role, I2S_ROLE_MASTER or I2S_ROLE_SLAVE

uint32_t **dma_desc_num**

I2S DMA buffer number, it is also the number of DMA descriptor

uint32_t **dma_frame_num**

I2S frame number in one DMA buffer. One frame means one-time sample data in all slots, it should be the multiple of '3' when the data bit width is 24.

bool **auto_clear**

Set to auto clear DMA TX buffer, i2s will always send zero automatically if no data to send

struct **i2s_chan_info_t**

I2S channel information.

Public Members

i2s_port_t **id**

I2S port id

i2s_role_t **role**

I2S role, I2S_ROLE_MASTER or I2S_ROLE_SLAVE

i2s_dir_t **dir**

I2S channel direction

i2s_comm_mode_t **mode**

I2S channel communication mode

i2s_chan_handle_t **pair_chan**

I2S pair channel handle in duplex mode, always NULL in simplex mode

Macros

I2S_CHANNEL_DEFAULT_CONFIG(i2s_num, i2s_role)

get default I2S property

I2S_GPIO_UNUSED

Used in i2s_gpio_config_t for signals which are not used

I2S Types

Header File

- components/driver/include/driver/i2s_types.h

Structures

struct **i2s_event_data_t**

Event structure used in I2S event queue.

Public Members

void ***data**

The pointer of DMA buffer that just finished sending or receiving for `on_recv` and `on_sent` callback NULL for `on_recv_q_ovf` and `on_send_q_ovf` callback

size_t **size**

The buffer size of DMA buffer when success to send or receive, also the buffer size that dropped when queue overflow. It is related to the dma_frame_num and data_bit_width, typically it is fixed when data_bit_width is not changed.

Type Definitions

typedef struct i2s_channel_obj_t ***i2s_chan_handle_t**

i2s channel object handle, the control unit of the i2s driver

typedef bool (***i2s_isr_callback_t**)(*i2s_chan_handle_t* handle, *i2s_event_data_t* *event, void *user_ctx)

I2S event callback.

Param handle **[in]** I2S channel handle, created from `i2s_new_channel()`
Param event **[in]** I2S event data
Param user_ctx **[in]** User registered context, passed from `i2s_channel_register_event_callback()`
Return Whether a high priority task has been waken up by this callback function

Enumerations

enum **i2s_port_t**

I2S controller port number, the max port number is (SOC_I2S_NUM -1).

Values:

enumerator **I2S_NUM_0**

I2S controller port 0

enumerator **I2S_NUM_1**

I2S controller port 1

enumerator **I2S_NUM_AUTO**

Select whichever port is available

enum **i2s_comm_mode_t**

I2S controller communication mode.

Values:

enumerator **I2S_COMM_MODE_STD**

I2S controller using standard communication mode, support philips/MSB/PCM format

enumerator **I2S_COMM_MODE_PDM**

I2S controller using PDM communication mode, support PDM output or input

enumerator **I2S_COMM_MODE_NONE**

Unspecified I2S controller mode

enum **i2s_mclk_multiple_t**

The multiple of mclk to sample rate.

Values:

enumerator **I2S_MCLK_MULTIPLE_128**

mclk = sample_rate * 128

enumerator **I2S_MCLK_MULTIPLE_256**

mclk = sample_rate * 256

enumerator **I2S_MCLK_MULTIPLE_384**

mclk = sample_rate * 384

enumerator **I2S_MCLK_MULTIPLE_512**

mclk = sample_rate * 512

Header File

- components/hal/include/hal/i2s_types.h

Type Definitions

typedef *soc_periph_i2s_clk_src_t* **i2s_clock_src_t**

I2S clock source

Enumerations

enum **i2s_slot_mode_t**

I2S channel slot mode.

Values:

enumerator **I2S_SLOT_MODE_MONO**

I2S channel slot format mono, transmit same data in all slots for tx mode, only receive the data in the first slots for rx mode.

enumerator **I2S_SLOT_MODE_STEREO**

I2S channel slot format stereo, transmit different data in different slots for tx mode, receive the data in all slots for rx mode.

enum **i2s_dir_t**

I2S channel direction.

Values:

enumerator **I2S_DIR_RX**

I2S channel direction RX

enumerator **I2S_DIR_TX**

I2S channel direction TX

enum **i2s_role_t**

I2S controller role.

Values:

enumerator **I2S_ROLE_MASTER**

I2S controller master role, bclk and ws signal will be set to output

enumerator **I2S_ROLE_SLAVE**

I2S controller slave role, bclk and ws signal will be set to input

enum **i2s_data_bit_width_t**

Available data bit width in one slot.

Values:

enumerator **I2S_DATA_BIT_WIDTH_8BIT**

I2S channel data bit-width: 8

enumerator **I2S_DATA_BIT_WIDTH_16BIT**

I2S channel data bit-width: 16

enumerator **I2S_DATA_BIT_WIDTH_24BIT**

I2S channel data bit-width: 24

enumerator **I2S_DATA_BIT_WIDTH_32BIT**

I2S channel data bit-width: 32

enum **i2s_slot_bit_width_t**

Total slot bit width in one slot.

Values:

enumerator **I2S_SLOT_BIT_WIDTH_AUTO**

I2S channel slot bit-width equals to data bit-width

enumerator **I2S_SLOT_BIT_WIDTH_8BIT**

I2S channel slot bit-width: 8

enumerator **I2S_SLOT_BIT_WIDTH_16BIT**

I2S channel slot bit-width: 16

enumerator **I2S_SLOT_BIT_WIDTH_24BIT**

I2S channel slot bit-width: 24

enumerator **I2S_SLOT_BIT_WIDTH_32BIT**

I2S channel slot bit-width: 32

enum **i2s_pdm_dsr_t**

I2S PDM RX down-sampling mode.

Values:

enumerator **I2S_PDM_DSR_8S**

downsampling number is 8 for PDM RX mode

enumerator **I2S_PDM_DSR_16S**

downsampling number is 16 for PDM RX mode

enumerator **I2S_PDM_DSR_MAX**

enum **i2s_pdm_sig_scale_t**

pdm tx singnal scaling mode

Values:

enumerator **I2S_PDM_SIG_SCALING_DIV_2**

I2S TX PDM signal scaling: /2

enumerator **I2S_PDM_SIG_SCALING_MUL_1**

I2S TX PDM signal scaling: x1

enumerator **I2S_PDM_SIG_SCALING_MUL_2**

I2S TX PDM signal scaling: x2

enumerator **I2S_PDM_SIG_SCALING_MUL_4**

I2S TX PDM signal scaling: x4

enum **i2s_std_slot_mask_t**

I2S slot select in standard mode.

Note: It has different meanings in tx/rx/mono/stereo mode, and it may have differen behaviors on different targets For the details, please refer to the I2S API reference

---

Values:

enumerator **I2S_STD_SLOT_LEFT**

I2S transmits or receives left slot

enumerator **I2S_STD_SLOT_RIGHT**

I2S transmits or receives right slot

enumerator **I2S_STD_SLOT_BOTH**

I2S transmits or receives both left and right slot

enum **i2s_pdm_slot_mask_t**

I2S slot select in PDM mode.

Values:

enumerator **I2S_PDM_SLOT_RIGHT**

I2S PDM only transmits or receives the PDM device whose ‘select’ pin is pulled up

enumerator **I2S_PDM_SLOT_LEFT**

I2S PDM only transmits or receives the PDM device whose ‘select’ pin is pulled down

enumerator **I2S_PDM_SLOT_BOTH**

I2S PDM transmits or receives both two slots

2.6.10 LCD

Introduction

ESP chips can generate various kinds of timings that needed by common LCDs on the market, like SPI LCD, I80 LCD (a.k.a Intel 8080 parallel LCD), RGB/SRGB LCD, I2C LCD, etc. The `esp_lcd` component is officially to support those LCDs with a group of universal APIs across chips.

Functional Overview

In `esp_lcd`, an LCD panel is represented by *`esp_lcd_panel_handle_t`*, which plays the role of an **abstract frame buffer**, regardless of the frame memory is allocated inside ESP chip or in external LCD controller. Based on the location of the frame buffer and the hardware connection interface, the LCD panel drivers are mainly grouped into the following categories:

- Controller based LCD driver involves multiple steps to get a panel handle, like bus allocation, IO device registration and controller driver install. The frame buffer is located in the controller's internal GRAM (Graphical RAM). ESP-IDF provides only a limited number of LCD controller drivers out of the box (e.g., ST7789, SSD1306), *More Controller Based LCD Drivers* are maintained in the Espressif Component Registry.
- *SPI Interfaced LCD* describes the steps to install the SPI LCD IO driver and then get the panel handle.
- *I2C Interfaced LCD* describes the steps to install the I2C LCD IO driver and then get the panel handle.
- *I80 Interfaced LCD* describes the steps to install the I80 LCD IO driver and then get the panel handle.
- *LCD Panel IO Operations* - provides a set of APIs to operate the LCD panel, like turning on/off the display, setting the orientation, etc. These operations are common for either controller-based LCD panel driver or RGB LCD panel driver.

SPI Interfaced LCD

1. Create an SPI bus. Please refer to *SPI Master API doc* for more details.

```
spi_bus_config_t buscfg = {
    .sclk_io_num = EXAMPLE_PIN_NUM_SCLK,
    .mosi_io_num = EXAMPLE_PIN_NUM_MOSI,
    .miso_io_num = EXAMPLE_PIN_NUM_MISO,
    .quadwp_io_num = -1, // Quad SPI LCD driver is not yet supported
    .quadhd_io_num = -1, // Quad SPI LCD driver is not yet supported
    .max_transfer_sz = EXAMPLE_LCD_H_RES * 80 * sizeof(uint16_t), //␣
↪transfer 80 lines of pixels (assume pixel is RGB565) at most in one␣
↪SPI transaction
};
ESP_ERROR_CHECK(spi_bus_initialize(LCD_HOST, &buscfg, SPI_DMA_CH_
↪AUTO)); // Enable the DMA feature
```

2. Allocate an LCD IO device handle from the SPI bus. In this step, you need to provide the following information:
   - *`esp_lcd_panel_io_spi_config_t::dc_gpio_num`*: Sets the gpio number for the DC signal line (some LCD calls this `RS` line). The LCD driver will use this GPIO to switch between sending command and sending data.
   - *`esp_lcd_panel_io_spi_config_t::cs_gpio_num`*: Sets the gpio number for the CS signal line. The LCD driver will use this GPIO to select the LCD chip. If the SPI bus only has one device attached (i.e. this LCD), you can set the gpio number to `-1` to occupy the bus exclusively.
   - *`esp_lcd_panel_io_spi_config_t::pclk_hz`* sets the frequency of the pixel clock, in Hz. The value should not exceed the range recommended in the LCD spec.
   - *`esp_lcd_panel_io_spi_config_t::spi_mode`* sets the SPI mode. The LCD driver will use this mode to communicate with the LCD. For the meaning of the SPI mode, please refer to the *SPI Master API doc*.

- *esp_lcd_panel_io_spi_config_t::lcd_cmd_bits* and *esp_lcd_panel_io_spi_config_t::lcd_param_bits* set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.
- *esp_lcd_panel_io_spi_config_t::trans_queue_depth* sets the depth of the SPI transaction queue. A bigger value means more transactions can be queued up, but it also consumes more memory.

```
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_spi_config_t io_config = {
    .dc_gpio_num = EXAMPLE_PIN_NUM_LCD_DC,
    .cs_gpio_num = EXAMPLE_PIN_NUM_LCD_CS,
    .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_PARAM_BITS,
    .spi_mode = 0,
    .trans_queue_depth = 10,
};
// Attach the LCD to the SPI bus
ESP_ERROR_CHECK(esp_lcd_new_panel_io_spi((esp_lcd_spi_bus_handle_t)LCD_
↪HOST, &io_config, &io_handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the SPI IO device handle that allocated in the last step, and some panel specific configurations:
    - *esp_lcd_panel_dev_config_t::reset_gpio_num* sets the LCD's hardware reset GPIO number. If the LCD does not have a hardware reset pin, set this to `-1`.
    - *esp_lcd_panel_dev_config_t::rgb_ele_order* sets the R-G-B element order of each color data.
    - *esp_lcd_panel_dev_config_t::bits_per_pixel* sets the bit width of the pixel color data. The LCD driver uses this value to calculate the number of bytes to send to the LCD controller chip.
    - *esp_lcd_panel_dev_config_t::data_endian* specifies the data endian to be transmitted to the screen. No need to specify for color data within 1 byte, like RGB232. For drivers that do not support specifying data endian, this field would be ignored.

```
esp_lcd_panel_handle_t panel_handle = NULL;
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_BGR,
    .bits_per_pixel = 16,
};
// Create LCD panel handle for ST7789, with the SPI IO device handle
ESP_ERROR_CHECK(esp_lcd_new_panel_st7789(io_handle, &panel_config, &
↪panel_handle));
```

I2C Interfaced LCD

1. Create I2C bus. Please refer to *I2C API doc* for more details.

```
i2c_config_t i2c_conf = {
    .mode = I2C_MODE_MASTER, // I2C LCD is a master node
    .sda_io_num = EXAMPLE_PIN_NUM_SDA,
    .scl_io_num = EXAMPLE_PIN_NUM_SCL,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
};
ESP_ERROR_CHECK(i2c_param_config(I2C_HOST, &i2c_conf));
ESP_ERROR_CHECK(i2c_driver_install(I2C_HOST, I2C_MODE_MASTER, 0, 0,␣
↪0));
```

(continues on next page)

(continued from previous page)

2. Allocate an LCD IO device handle from the I2C bus. In this step, you need to provide the following information:
    - *esp_lcd_panel_io_i2c_config_t::dev_addr* sets the I2C device address of the LCD controller chip. The LCD driver will use this address to communicate with the LCD controller chip.
    - *esp_lcd_panel_io_i2c_config_t::lcd_cmd_bits* and *esp_lcd_panel_io_i2c_config_t::lcd_param_bits* set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.

```
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_i2c_config_t io_config = {
    .dev_addr = EXAMPLE_I2C_HW_ADDR,
    .control_phase_bytes = 1, // refer to LCD spec
    .dc_bit_offset = 6,       // refer to LCD spec
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_CMD_BITS,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_io_i2c((esp_lcd_i2c_bus_handle_t)I2C_
↪HOST, &io_config, &io_handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the I2C IO device handle that allocated in the last step, and some panel specific configurations:
    - *esp_lcd_panel_dev_config_t::reset_gpio_num* sets the LCD's hardware reset GPIO number. If the LCD does not have a hardware reset pin, set this to `-1`.
    - *esp_lcd_panel_dev_config_t::bits_per_pixel* sets the bit width of the pixel color data. The LCD driver will use this value to calculate the number of bytes to send to the LCD controller chip.

```
esp_lcd_panel_handle_t panel_handle = NULL;
esp_lcd_panel_dev_config_t panel_config = {
    .bits_per_pixel = 1,
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_ssd1306(io_handle, &panel_config, &
↪panel_handle));
```

I80 Interfaced LCD

1. Create I80 bus by *esp_lcd_new_i80_bus()*. You need to set up the following parameters for an Intel 8080 parallel bus:
    - *esp_lcd_i80_bus_config_t::clk_src* sets the clock source of the I80 bus. Note, the default clock source may be different between ESP targets.
    - *esp_lcd_i80_bus_config_t::wr_gpio_num* sets the GPIO number of the pixel clock (also referred as `WR` in some LCD spec)
    - *esp_lcd_i80_bus_config_t::dc_gpio_num* sets the GPIO number of the data/command select pin (also referred as `RS` in some LCD spec)
    - *esp_lcd_i80_bus_config_t::bus_width* sets the bit width of the data bus (only support `8` or `16`)
    - *esp_lcd_i80_bus_config_t::data_gpio_nums* is the array of the GPIO number of the data bus. The number of GPIOs should be equal to the *esp_lcd_i80_bus_config_t::bus_width* value.
    - *esp_lcd_i80_bus_config_t::max_transfer_bytes* sets the maximum number of bytes that can be transferred in one transaction.

```
esp_lcd_i80_bus_handle_t i80_bus = NULL;
esp_lcd_i80_bus_config_t bus_config = {
```

(continues on next page)

(continued from previous page)

```
    .clk_src = LCD_CLK_SRC_DEFAULT,
    .dc_gpio_num = EXAMPLE_PIN_NUM_DC,
    .wr_gpio_num = EXAMPLE_PIN_NUM_PCLK,
    .data_gpio_nums = {
        EXAMPLE_PIN_NUM_DATA0,
        EXAMPLE_PIN_NUM_DATA1,
        EXAMPLE_PIN_NUM_DATA2,
        EXAMPLE_PIN_NUM_DATA3,
        EXAMPLE_PIN_NUM_DATA4,
        EXAMPLE_PIN_NUM_DATA5,
        EXAMPLE_PIN_NUM_DATA6,
        EXAMPLE_PIN_NUM_DATA7,
    },
    .bus_width = 8,
    .max_transfer_bytes = EXAMPLE_LCD_H_RES * 100 * sizeof(uint16_t), /
↪/ transfer 100 lines of pixels (assume pixel is RGB565) at most in␣
↪one transaction
    .psram_trans_align = EXAMPLE_PSRAM_DATA_ALIGNMENT,
    .sram_trans_align = 4,
};
ESP_ERROR_CHECK(esp_lcd_new_i80_bus(&bus_config, &i80_bus));
```

2. Allocate an LCD IO device handle from the I80 bus. In this step, you need to provide the following information:
    - *esp_lcd_panel_io_i80_config_t::cs_gpio_num* sets the GPIO number of the chip select pin.
    - *esp_lcd_panel_io_i80_config_t::pclk_hz* sets the pixel clock frequency in Hz. Higher pixel clock frequency will result in higher refresh rate, but may cause flickering if the DMA bandwidth is not sufficient or the LCD controller chip does not support high pixel clock frequency.
    - *esp_lcd_panel_io_i80_config_t::lcd_cmd_bits* and *esp_lcd_panel_io_i80_config_t::lcd_param_bits* set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.
    - *esp_lcd_panel_io_i80_config_t::trans_queue_depth* sets the maximum number of transactions that can be queued in the LCD IO device. A bigger value means more transactions can be queued up, but it also consumes more memory.

```
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_i80_config_t io_config = {
    .cs_gpio_num = EXAMPLE_PIN_NUM_CS,
    .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
    .trans_queue_depth = 10,
    .dc_levels = {
        .dc_idle_level = 0,
        .dc_cmd_level = 0,
        .dc_dummy_level = 0,
        .dc_data_level = 1,
    },
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_PARAM_BITS,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_io_i80(i80_bus, &io_config, &io_
↪handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the I80 IO device handle that allocated in the last step, and some panel specific configurations:
    - *esp_lcd_panel_dev_config_t::bits_per_pixel* sets the bit width of the pixel color data. The LCD driver will use this value to calculate the number of bytes to send to the LCD controller chip.
    - *esp_lcd_panel_dev_config_t::reset_gpio_num* sets the GPIO number of the

reset pin. If the LCD controller chip does not have a reset pin, you can set this value to `-1`.
- *esp_lcd_panel_dev_config_t::rgb_ele_order* sets the color order the pixel color data.

```
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB,
    .bits_per_pixel = 16,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_st7789(io_handle, &panel_config, &
↪panel_handle));
```

More Controller Based LCD Drivers

More LCD panel drivers and touch drivers are available in IDF Component Registry. The list of available and planned drivers with links is in this table.

LCD Panel IO Operations

- *esp_lcd_panel_reset()* can reset the LCD panel.
- Use *esp_lcd_panel_swap_xy()* and *esp_lcd_panel_mirror()*, you can rotate the LCD screen.
- *esp_lcd_panel_disp_on_off()* can turn on or off the LCD screen (different from LCD backlight).
- *esp_lcd_panel_draw_bitmap()* is the most significant function, that will do the magic to draw the user provided color buffer to the LCD screen, where the draw window is also configurable.

Application Example

LCD examples are located under: peripherals/lcd:

- Universal SPI LCD example with SPI touch - peripherals/lcd/spi_lcd_touch
- Jpeg decoding and LCD display - peripherals/lcd/tjpgd
- i80 controller based LCD and LVGL animation UI - peripherals/lcd/i80_controller
- RGB panel example with scatter chart UI - peripherals/lcd/rgb_panel
- I2C interfaced OLED display scrolling text - peripherals/lcd/i2c_oled

API Reference

Header File

- components/hal/include/hal/lcd_types.h

Macros

LCD_RGB_ENDIAN_RGB

LCD_RGB_ENDIAN_BGR

Type Definitions

typedef *soc_periph_lcd_clk_src_t* **lcd_clock_source_t**

LCD clock source.

typedef *lcd_rgb_element_order_t* **lcd_color_rgb_endian_t**

for backward compatible

Enumerations

enum **lcd_rgb_element_order_t**

RGB color endian.

Values:

enumerator **LCD_RGB_ELEMENT_ORDER_RGB**

RGB element order: RGB

enumerator **LCD_RGB_ELEMENT_ORDER_BGR**

RGB element order: BGR

enum **lcd_rgb_data_endian_t**

RGB data endian.

Values:

enumerator **LCD_RGB_DATA_ENDIAN_BIG**

RGB data endian: MSB first

enumerator **LCD_RGB_DATA_ENDIAN_LITTLE**

RGB data endian: LSB first

enum **lcd_color_space_t**

LCD color space.

Values:

enumerator **LCD_COLOR_SPACE_RGB**

Color space: RGB

enumerator **LCD_COLOR_SPACE_YUV**

Color space: YUV

enum **lcd_color_range_t**

LCD color range.

Values:

enumerator **LCD_COLOR_RANGE_LIMIT**

Limited color range

enumerator **LCD_COLOR_RANGE_FULL**

Full color range

enum **lcd_yuv_sample_t**

YUV sampling method.

Values:

enumerator **LCD_YUV_SAMPLE_422**

YUV 4:2:2 sampling

enumerator **LCD_YUV_SAMPLE_420**

YUV 4:2:0 sampling

enumerator **LCD_YUV_SAMPLE_411**

YUV 4:1:1 sampling

enum **lcd_yuv_conv_std_t**

The standard used for conversion between RGB and YUV.

Values:

enumerator **LCD_YUV_CONV_STD_BT601**

YUV<->RGB conversion standard: BT.601

enumerator **LCD_YUV_CONV_STD_BT709**

YUV<->RGB conversion standard: BT.709

Header File

- components/esp_lcd/include/esp_lcd_types.h

Type Definitions

typedef struct esp_lcd_panel_io_t ***esp_lcd_panel_io_handle_t**

Type of LCD panel IO handle

typedef struct esp_lcd_panel_t ***esp_lcd_panel_handle_t**

Type of LCD panel handle

Header File

- components/esp_lcd/include/esp_lcd_panel_io.h

Functions

esp_err_t **esp_lcd_panel_io_rx_param**(*esp_lcd_panel_io_handle_t* io, int lcd_cmd, void *param, size_t param_size)

Transmit LCD command and receive corresponding parameters.

Note: Commands sent by this function are short, so they are sent using polling transactions. The function does not return before the command transfer is completed. If any queued transactions sent by `esp_lcd_panel_io_tx_color()` are still pending when this function is called, this function will wait until they are finished and the queue is empty before sending the command(s).

Parameters

- **io** –**[in]** LCD panel IO handle, which is created by other factory API like `esp_lcd_new_panel_io_spi()`
- **lcd_cmd** –**[in]** The specific LCD command, set to -1 if no command needed
- **param** –**[out]** Buffer for the command data
- **param_size** –**[in]** Size of `param` buffer

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_SUPPORTED if read is not supported by transport

- ESP_OK on success

esp_err_t **esp_lcd_panel_io_tx_param**(*esp_lcd_panel_io_handle_t* io, int lcd_cmd, const void *param, size_t param_size)

Transmit LCD command and corresponding parameters.

Note: Commands sent by this function are short, so they are sent using polling transactions. The function does not return before the command transfer is completed. If any queued transactions sent by `esp_lcd_panel_io_tx_color()` are still pending when this function is called, this function will wait until they are finished and the queue is empty before sending the command(s).

Parameters

- **io** –**[in]** LCD panel IO handle, which is created by other factory API like `esp_lcd_new_panel_io_spi()`
- **lcd_cmd** –**[in]** The specific LCD command, set to -1 if no command needed
- **param** –**[in]** Buffer that holds the command specific parameters, set to NULL if no parameter is needed for the command
- **param_size** –**[in]** Size of `param` in memory, in bytes, set to zero if no parameter is needed for the command

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **esp_lcd_panel_io_tx_color**(*esp_lcd_panel_io_handle_t* io, int lcd_cmd, const void *color, size_t color_size)

Transmit LCD RGB data.

Note: This function will package the command and RGB data into a transaction, and push into a queue. The real transmission is performed in the background (DMA+interrupt). The caller should take care of the lifecycle of the `color` buffer. Recycling of color buffer should be done in the callback `on_color_trans_done()`.

Parameters

- **io** –**[in]** LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`
- **lcd_cmd** –**[in]** The specific LCD command, set to -1 if no command needed
- **color** –**[in]** Buffer that holds the RGB color data
- **color_size** –**[in]** Size of `color` in memory, in bytes

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **esp_lcd_panel_io_del**(*esp_lcd_panel_io_handle_t* io)

Destroy LCD panel IO handle (deinitialize panel and free all corresponding resource)

Parameters **io** –**[in]** LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **esp_lcd_panel_io_register_event_callbacks**(*esp_lcd_panel_io_handle_t* io, const *esp_lcd_panel_io_callbacks_t* *cbs, void *user_ctx)

Register LCD panel IO callbacks.

Parameters

- **io** –**[in]** LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`
- **cbs** –**[in]** structure with all LCD panel IO callbacks
- **user_ctx** –**[in]** User private data, passed directly to callback’ s user_ctx

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **esp_lcd_new_panel_io_spi**(*esp_lcd_spi_bus_handle_t* bus, const *esp_lcd_panel_io_spi_config_t* *io_config, *esp_lcd_panel_io_handle_t* *ret_io)

Create LCD panel IO handle, for SPI interface.

Parameters

- **bus** –**[in]** SPI bus handle
- **io_config** –**[in]** IO configuration, for SPI interface
- **ret_io** –**[out]** Returned IO handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **esp_lcd_new_panel_io_i2c**(*esp_lcd_i2c_bus_handle_t* bus, const *esp_lcd_panel_io_i2c_config_t* *io_config, *esp_lcd_panel_io_handle_t* *ret_io)

Create LCD panel IO handle, for I2C interface.

Parameters

- **bus** –**[in]** I2C bus handle
- **io_config** –**[in]** IO configuration, for I2C interface
- **ret_io** –**[out]** Returned IO handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **esp_lcd_new_i80_bus**(const *esp_lcd_i80_bus_config_t* *bus_config, *esp_lcd_i80_bus_handle_t* *ret_bus)

Create Intel 8080 bus handle.

Parameters

- **bus_config** –**[in]** Bus configuration
- **ret_bus** –**[out]** Returned bus handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_NOT_FOUND if no free bus is available
- ESP_OK on success

esp_err_t **esp_lcd_del_i80_bus**(*esp_lcd_i80_bus_handle_t* bus)

Destroy Intel 8080 bus handle.

Parameters **bus** –**[in]** Intel 8080 bus handle, created by `esp_lcd_new_i80_bus()`

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if there still be some device attached to the bus
- ESP_OK on success

esp_err_t **esp_lcd_new_panel_io_i80**(*esp_lcd_i80_bus_handle_t* bus, const *esp_lcd_panel_io_i80_config_t* *io_config, *esp_lcd_panel_io_handle_t* *ret_io)

Create LCD panel IO, for Intel 8080 interface.

Parameters

- **bus** –**[in]** Intel 8080 bus handle, created by `esp_lcd_new_i80_bus()`
- **io_config** –**[in]** IO configuration, for i80 interface
- **ret_io** –**[out]** Returned panel IO handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_SUPPORTED if some configuration can't be satisfied, e.g. pixel clock out of the range
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Structures

struct **esp_lcd_panel_io_event_data_t**

Type of LCD panel IO event data.

struct **esp_lcd_panel_io_callbacks_t**

Type of LCD panel IO callbacks.

Public Members

esp_lcd_panel_io_color_trans_done_cb_t **on_color_trans_done**

Callback invoked when color data transfer has finished

struct **esp_lcd_panel_io_spi_config_t**

Panel IO configuration structure, for SPI interface.

Public Members

int **cs_gpio_num**

GPIO used for CS line

int **dc_gpio_num**

GPIO used to select the D/C line, set this to -1 if the D/C line is not used

int **spi_mode**

Traditional SPI mode (0~3)

unsigned int **pclk_hz**

Frequency of pixel clock

size_t **trans_queue_depth**

Size of internal transaction queue

esp_lcd_panel_io_color_trans_done_cb_t **on_color_trans_done**

Callback invoked when color data transfer has finished

void ***user_ctx**

User private data, passed directly to on_color_trans_done's user_ctx

int **lcd_cmd_bits**

Bit-width of LCD command

int **lcd_param_bits**

Bit-width of LCD parameter

unsigned int **dc_high_on_cmd**

If enabled, DC level = 1 indicates command transfer

unsigned int **dc_low_on_data**

If enabled, DC level = 0 indicates color data transfer

unsigned int **dc_low_on_param**

If enabled, DC level = 0 indicates parameter transfer

unsigned int **octal_mode**

transmit with octal mode (8 data lines), this mode is used to simulate Intel 8080 timing

unsigned int **quad_mode**

transmit with quad mode (4 data lines), this mode is useful when transmitting LCD parameters (Only use one line for command)

unsigned int **sio_mode**

Read and write through a single data line (MOSI)

unsigned int **lsb_first**

transmit LSB bit first

unsigned int **cs_high_active**

CS line is high active

struct *esp_lcd_panel_io_spi_config_t*::[anonymous] **flags**

Extra flags to fine-tune the SPI device

struct **esp_lcd_panel_io_i2c_config_t**

Panel IO configuration structure, for I2C interface.

Public Members

uint32_t **dev_addr**

I2C device address

esp_lcd_panel_io_color_trans_done_cb_t **on_color_trans_done**

Callback invoked when color data transfer has finished

void ***user_ctx**

User private data, passed directly to on_color_trans_done’s user_ctx

size_t **control_phase_bytes**

I2C LCD panel will encode control information (e.g. D/C selection) into control phase, in several bytes

unsigned int **dc_bit_offset**

Offset of the D/C selection bit in control phase

int **lcd_cmd_bits**

Bit-width of LCD command

int **lcd_param_bits**

Bit-width of LCD parameter

unsigned int **dc_low_on_data**

If this flag is enabled, DC line = 0 means transfer data, DC line = 1 means transfer command; vice versa

unsigned int **disable_control_phase**

If this flag is enabled, the control phase isn't used

struct *esp_lcd_panel_io_i2c_config_t*::[anonymous] **flags**

Extra flags to fine-tune the I2C device

struct **esp_lcd_i80_bus_config_t**

LCD Intel 8080 bus configuration structure.

Public Members

int **dc_gpio_num**

GPIO used for D/C line

int **wr_gpio_num**

GPIO used for WR line

lcd_clock_source_t **clk_src**

Clock source for the I80 LCD peripheral

int **data_gpio_nums**[(24)]

GPIOs used for data lines

size_t **bus_width**

Number of data lines, 8 or 16

size_t **max_transfer_bytes**

Maximum transfer size, this determines the length of internal DMA link

size_t **psram_trans_align**

DMA transfer alignment for data allocated from PSRAM

size_t **sram_trans_align**

DMA transfer alignment for data allocated from SRAM

struct **esp_lcd_panel_io_i80_config_t**

Panel IO configuration structure, for intel 8080 interface.

Public Members

int **cs_gpio_num**

GPIO used for CS line, set to -1 will declaim exclusively use of I80 bus

uint32_t **pclk_hz**

Frequency of pixel clock

size_t **trans_queue_depth**

Transaction queue size, larger queue, higher throughput

esp_lcd_panel_io_color_trans_done_cb_t **on_color_trans_done**

Callback invoked when color data was transferred done

void ***user_ctx**

User private data, passed directly to on_color_trans_done’ s user_ctx

int **lcd_cmd_bits**

Bit-width of LCD command

int **lcd_param_bits**

Bit-width of LCD parameter

unsigned int **dc_idle_level**

Level of DC line in IDLE phase

unsigned int **dc_cmd_level**

Level of DC line in CMD phase

unsigned int **dc_dummy_level**

Level of DC line in DUMMY phase

unsigned int **dc_data_level**

Level of DC line in DATA phase

struct *esp_lcd_panel_io_i80_config_t*::[anonymous] **dc_levels**

Each i80 device might have its own D/C control logic

unsigned int **cs_active_high**

If set, a high level of CS line will select the device, otherwise, CS line is low level active

unsigned int **reverse_color_bits**

Reverse the data bits, D[N:0] -> D[0:N]

unsigned int **swap_color_bytes**

Swap adjacent two color bytes

unsigned int **pclk_active_neg**

The display will write data lines when there' s a falling edge on WR signal (a.k.a the PCLK)

unsigned int **pclk_idle_low**

The WR signal (a.k.a the PCLK) stays at low level in IDLE phase

struct *esp_lcd_panel_io_i80_config_t*::[anonymous] **flags**

Panel IO config flags

Type Definitions

typedef void ***esp_lcd_spi_bus_handle_t**

Type of LCD SPI bus handle

typedef void ***esp_lcd_i2c_bus_handle_t**

Type of LCD I2C bus handle

typedef struct esp_lcd_i80_bus_t ***esp_lcd_i80_bus_handle_t**

Type of LCD intel 8080 bus handle

typedef bool (***esp_lcd_panel_io_color_trans_done_cb_t**)(*esp_lcd_panel_io_handle_t* panel_io, *esp_lcd_panel_io_event_data_t* *edata, void *user_ctx)

Declare the prototype of the function that will be invoked when panel IO finishes transferring color data.

Param panel_io [in] LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`
Param edata [in] Panel IO event data, fed by driver
Param user_ctx [in] User data, passed from `esp_lcd_panel_io_xxx_config_t`
Return Whether a high priority task has been waken up by this function

Header File

- components/esp_lcd/include/esp_lcd_panel_ops.h

Functions

esp_err_t **esp_lcd_panel_reset**(*esp_lcd_panel_handle_t* panel)

Reset LCD panel.

Note: Panel reset must be called before attempting to initialize the panel using `esp_lcd_panel_init()`.

Parameters **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_init**(*esp_lcd_panel_handle_t* panel)

Initialize LCD panel.

Note: Before calling this function, make sure the LCD panel has finished the `reset` stage by `esp_lcd_panel_reset()`.

Parameters **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`

Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_del**(*esp_lcd_panel_handle_t* panel)

Deinitialize the LCD panel.

Parameters **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`

Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_draw_bitmap**(*esp_lcd_panel_handle_t* panel, int x_start, int y_start, int x_end, int y_end, const void *color_data)

Draw bitmap on LCD panel.

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **x_start** –**[in]** Start index on x-axis (x_start included)
- **y_start** –**[in]** Start index on y-axis (y_start included)
- **x_end** –**[in]** End index on x-axis (x_end not included)
- **y_end** –**[in]** End index on y-axis (y_end not included)
- **color_data** –**[in]** RGB color data that will be dumped to the specific window range

Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_mirror**(*esp_lcd_panel_handle_t* panel, bool mirror_x, bool mirror_y)

Mirror the LCD panel on specific axis.

Note: Combined with `esp_lcd_panel_swap_xy()`, one can realize screen rotation

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **mirror_x** –**[in]** Whether the panel will be mirrored about the x axis
- **mirror_y** –**[in]** Whether the panel will be mirrored about the y axis

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

esp_err_t **esp_lcd_panel_swap_xy**(*esp_lcd_panel_handle_t* panel, bool swap_axes)

Swap/Exchange x and y axis.

Note: Combined with `esp_lcd_panel_mirror()`, one can realize screen rotation

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **swap_axes** –**[in]** Whether to swap the x and y axis

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

esp_err_t **esp_lcd_panel_set_gap**(*esp_lcd_panel_handle_t* panel, int x_gap, int y_gap)

Set extra gap in x and y axis.

The gap is the space (in pixels) between the left/top sides of the LCD panel and the first row/column respectively of the actual contents displayed.

Note: Setting a gap is useful when positioning or centering a frame that is smaller than the LCD.

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **x_gap** –**[in]** Extra gap on x axis, in pixels
- **y_gap** –**[in]** Extra gap on y axis, in pixels

Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_invert_color**(*esp_lcd_panel_handle_t* panel, bool invert_color_data)

Invert the color (bit-wise invert the color data line)

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **invert_color_data** –**[in]** Whether to invert the color data

Returns

- ESP_OK on success

esp_err_t **esp_lcd_panel_disp_on_off**(*esp_lcd_panel_handle_t* panel, bool on_off)

Turn on or off the display.

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **on_off** –**[in]** True to turns on display, False to turns off display

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

esp_err_t **esp_lcd_panel_disp_off**(*esp_lcd_panel_handle_t* panel, bool off)

Turn off the display.

Parameters

- **panel** –**[in]** LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **off** –**[in]** Whether to turn off the screen

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

Header File

- components/esp_lcd/include/esp_lcd_panel_rgb.h

Header File

- components/esp_lcd/include/esp_lcd_panel_vendor.h

Functions

esp_err_t **esp_lcd_new_panel_st7789**(const *esp_lcd_panel_io_handle_t* io, const *esp_lcd_panel_dev_config_t* *panel_dev_config, *esp_lcd_panel_handle_t* *ret_panel)

Create LCD panel for model ST7789.

Parameters

- **io** –**[in]** LCD panel IO handle
- **panel_dev_config** –**[in]** general panel device configuration
- **ret_panel** –**[out]** Returned LCD panel handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **esp_lcd_new_panel_nt35510**(const *esp_lcd_panel_io_handle_t* io, const *esp_lcd_panel_dev_config_t* *panel_dev_config, *esp_lcd_panel_handle_t* *ret_panel)

Create LCD panel for model NT35510.

Parameters

- **io** –**[in]** LCD panel IO handle
- **panel_dev_config** –**[in]** general panel device configuration
- **ret_panel** –**[out]** Returned LCD panel handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **esp_lcd_new_panel_ssd1306**(const *esp_lcd_panel_io_handle_t* io, const *esp_lcd_panel_dev_config_t* *panel_dev_config, *esp_lcd_panel_handle_t* *ret_panel)

Create LCD panel for model SSD1306.

Parameters

- **io** –**[in]** LCD panel IO handle
- **panel_dev_config** –**[in]** general panel device configuration
- **ret_panel** –**[out]** Returned LCD panel handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Structures

struct **esp_lcd_panel_dev_config_t**

Configuration structure for panel device.

Public Members

int **reset_gpio_num**

GPIO used to reset the LCD panel, set to -1 if it' s not used

lcd_rgb_element_order_t **color_space**

Deprecated:

Set RGB color space, please use rgb_ele_order instead

lcd_rgb_element_order_t **rgb_endian**

Deprecated:

Set RGB data endian, please use rgb_ele_order instead

lcd_rgb_element_order_t **rgb_ele_order**

Set RGB element order, RGB or BGR

lcd_rgb_data_endian_t **data_endian**

Set the data endian for color data larger than 1 byte

unsigned int **bits_per_pixel**

Color depth, in bpp

unsigned int **reset_active_high**

Setting this if the panel reset is high level active

struct *esp_lcd_panel_dev_config_t*::[anonymous] **flags**

LCD panel config flags

void ***vendor_config**

vendor specific configuration, optional, left as NULL if not used

2.6.11 LED Control (LEDC)

Introduction

The LED control (LEDC) peripheral is primarily designed to control the intensity of LEDs, although it can also be used to generate PWM signals for other purposes. It has 16 channels which can generate independent waveforms that can be used, for example, to drive RGB LED devices.

LEDC channels are divided into two groups of 8 channels each. One group of LEDC channels operates in high speed mode. This mode is implemented in hardware and offers automatic and glitch-free changing of the PWM duty cycle. The other group of channels operate in low speed mode, the PWM duty cycle must be changed by the driver in software. Each group of channels is also able to use different clock sources.

The PWM controller can automatically increase or decrease the duty cycle gradually, allowing for fades without any processor interference.

Functionality Overview

Setting up a channel of the LEDC in either *high or low speed mode* is done in three steps:

1. *Timer Configuration* by specifying the PWM signal's frequency and duty cycle resolution.
2. *Channel Configuration* by associating it with the timer and GPIO to output the PWM signal.

3. *Change PWM Signal* that drives the output in order to change LED's intensity. This can be done under the full control of software or with hardware fading functions.

As an optional step, it is also possible to set up an interrupt on fade end.

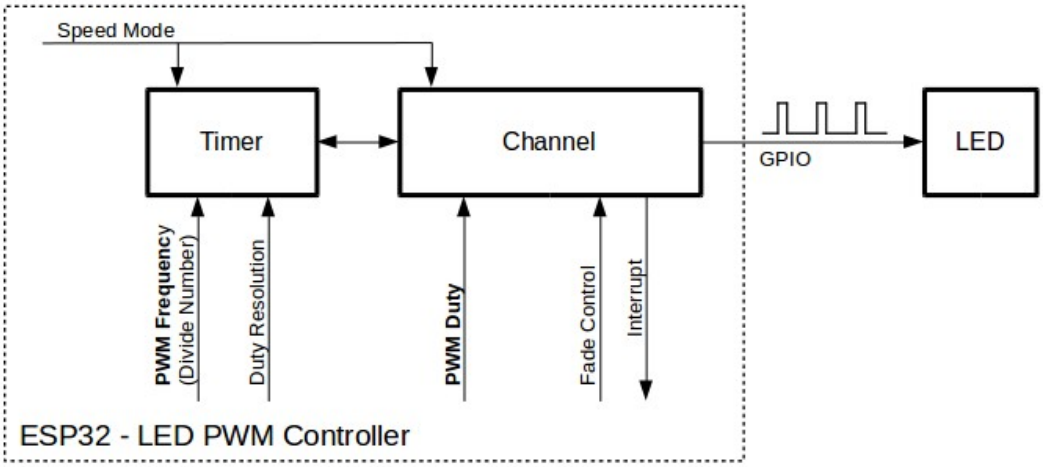


Fig. 11: Key Settings of LED PWM Controller's API

Timer Configuration Setting the timer is done by calling the function `ledc_timer_config()` and passing the data structure `ledc_timer_config_t` that contains the following configuration settings:

- Speed mode `ledc_mode_t`
- Timer number `ledc_timer_t`
- PWM signal frequency
- Resolution of PWM duty
- Source clock `ledc_clk_cfg_t`

The frequency and the duty resolution are interdependent. The higher the PWM frequency, the lower the duty resolution which is available, and vice versa. This relationship might be important if you are planning to use this API for purposes other than changing the intensity of LEDs. For more details, see Section *Supported Range of Frequency and Duty Resolutions*.

The source clock can also limit the PWM frequency. The higher the source clock frequency, the higher the maximum PWM frequency can be configured.

Table 5: Characteristics of ESP32 LEDC source clocks

Clock name	Clock freq	Speed mode	Clock capabilities
APB_CLK	80 MHz	High / Low	/
REF_TICK	1 MHz	High / Low	Dynamic Frequency Scaling compatible
RTC8M_CLK	~8 MHz	Low	Dynamic Frequency Scaling compatible, Light sleep compatible

Note:

1. On ESP32, if RTCxM_CLK is chosen as the LEDC clock source, an internal calibration will be performed to get the exact frequency of the clock. This ensures the accuracy of output PWM signal frequency.

Channel Configuration When the timer is set up, configure the desired channel (one out of *ledc_channel_t*). This is done by calling the function *ledc_channel_config()*.

Similar to the timer configuration, the channel setup function should be passed a structure *ledc_channel_config_t* that contains the channel's configuration parameters.

At this point, the channel should start operating and generating the PWM signal on the selected GPIO, as configured in *ledc_channel_config_t*, with the frequency specified in the timer settings and the given duty cycle. The channel operation (signal generation) can be suspended at any time by calling the function *ledc_stop()*.

Change PWM Signal Once the channel starts operating and generating the PWM signal with the constant duty cycle and frequency, there are a couple of ways to change this signal. When driving LEDs, primarily the duty cycle is changed to vary the light intensity.

The following two sections describe how to change the duty cycle using software and hardware fading. If required, the signal's frequency can also be changed; it is covered in Section *Change PWM Frequency*.

Change PWM Duty Cycle Using Software To set the duty cycle, use the dedicated function *ledc_set_duty()*. After that, call *ledc_update_duty()* to activate the changes. To check the currently set value, use the corresponding `_get_` function *ledc_get_duty()*.

Another way to set the duty cycle, as well as some other channel parameters, is by calling *ledc_channel_config()* covered in Section *Channel Configuration*.

The range of the duty cycle values passed to functions depends on selected `duty_resolution` and should be from `0` to `(2 ** duty_resolution)`. For example, if the selected duty resolution is 10, then the duty cycle values can range from 0 to 1024. This provides the resolution of ~ 0.1%.

Warning: On ESP32, when channel's binded timer selects its maximum duty resolution, the duty cycle value cannot be set to `(2 ** duty_resolution)`. Otherwise, the internal duty counter in the hardware will overflow and be messed up.

Change PWM Duty Cycle using Hardware The LEDC hardware provides the means to gradually transition from one duty cycle value to another. To use this functionality, enable fading with *ledc_fade_func_install()* and then configure it by calling one of the available fading functions:

- *ledc_set_fade_with_time()*
- *ledc_set_fade_with_step()*
- *ledc_set_fade()*

Start fading with *ledc_fade_start()*. A fade can be operated in blocking or non-blocking mode, please check *ledc_fade_mode_t* for the difference between the two available fade modes. Note that with either fade mode, the next fade or fixed-duty update will not take effect until the last fade finishes. Due to hardware limitations, there is no way to stop a fade before it reaches its target duty.

To get a notification about the completion of a fade operation, a fade end callback function can be registered for each channel by calling *ledc_cb_register()* after the fade service being installed. The fade end callback prototype is defined in *ledc_cb_t*, where you should return a boolean value from the callback function, indicating whether a high priority task is woken up by this callback function. It is worth mentioning, the callback and the function invoked by itself should be placed in IRAM, as the interrupt service routine is in IRAM. *ledc_cb_register()* will print a warning message if it finds the addresses of callback and user context are incorrect.

If not required anymore, fading and an associated interrupt can be disabled with *ledc_fade_func_uninstall()*.

Change PWM Frequency The LEDC API provides several ways to change the PWM frequency "on the fly":

- Set the frequency by calling *ledc_set_freq()*. There is a corresponding function *ledc_get_freq()* to check the current frequency.

- Change the frequency and the duty resolution by calling `ledc_bind_channel_timer()` to bind some other timer to the channel.
- Change the channel's timer by calling `ledc_channel_config()`.

More Control Over PWM There are several lower level timer-specific functions that can be used to change PWM settings:

- `ledc_timer_set()`
- `ledc_timer_rst()`
- `ledc_timer_pause()`
- `ledc_timer_resume()`

The first two functions are called "behind the scenes" by `ledc_channel_config()` to provide a startup of a timer after it is configured.

Use Interrupts When configuring an LEDC channel, one of the parameters selected within `ledc_channel_config_t` is `ledc_intr_type_t` which triggers an interrupt on fade completion.

For registration of a handler to address this interrupt, call `ledc_isr_register()`.

LEDC High and Low Speed Mode

High speed mode enables a glitch-free changeover of timer settings. This means that if the timer settings are modified, the changes will be applied automatically on the next overflow interrupt of the timer. In contrast, when updating the low-speed timer, the change of settings should be explicitly triggered by software. The LEDC driver handles it in the background, e.g., when `ledc_timer_config()` or `ledc_timer_set()` is called.

For additional details regarding speed modes, see *ESP32 Technical Reference Manual* > *LED PWM Controller (LEDC)* [PDF].

Supported Range of Frequency and Duty Resolutions

The LED PWM Controller is designed primarily to drive LEDs. It provides a large flexibility of PWM duty cycle settings. For instance, the PWM frequency of 5 kHz can have the maximum duty resolution of 13 bits. This means that the duty can be set anywhere from 0 to 100% with a resolution of ~0.012% (2 ** 13 = 8192 discrete levels of the LED intensity). Note, however, that these parameters depend on the clock signal clocking the LED PWM Controller timer which in turn clocks the channel (see *timer configuration* and the *ESP32 Technical Reference Manual* > *LED PWM Controller (LEDC)* [PDF]).

The LEDC can be used for generating signals at much higher frequencies that are sufficient enough to clock other devices, e.g., a digital camera module. In this case, the maximum available frequency is 40 MHz with duty resolution of 1 bit. This means that the duty cycle is fixed at 50% and cannot be adjusted.

The LEDC API is designed to report an error when trying to set a frequency and a duty resolution that exceed the range of LEDC's hardware. For example, an attempt to set the frequency to 20 MHz and the duty resolution to 3 bits will result in the following error reported on a serial monitor:

```
E (196) ledc: requested frequency and duty resolution cannot be achieved, try␣
↪reducing freq_hz or duty_resolution. div_param=128
```

In such a situation, either the duty resolution or the frequency must be reduced. For example, setting the duty resolution to 2 will resolve this issue and will make it possible to set the duty cycle at 25% steps, i.e., at 25%, 50% or 75%.

The LEDC driver will also capture and report attempts to configure frequency / duty resolution combinations that are below the supported minimum, e.g.:

```
E (196) ledc: requested frequency and duty resolution cannot be achieved, try␣
↪increasing freq_hz or duty_resolution. div_param=128000000
```

The duty resolution is normally set using *ledc_timer_bit_t*. This enumeration covers the range from 10 to 15 bits. If a smaller duty resolution is required (from 10 down to 1), enter the equivalent numeric values directly.

Application Example

The LEDC change duty cycle and fading control example: peripherals/ledc/ledc_fade.

The LEDC basic example: peripherals/ledc/ledc_basic.

API Reference

Header File

- components/driver/include/driver/ledc.h

Functions

esp_err_t **ledc_channel_config**(const *ledc_channel_config_t* *ledc_conf)

LEDC channel configuration Configure LEDC channel with the given channel/output gpio_num/interrupt/source timer/frequency(Hz)/LEDC duty resolution.

Parameters **ledc_conf** –Pointer of LEDC channel configure struct

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **ledc_timer_config**(const *ledc_timer_config_t* *timer_conf)

LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution.

Parameters **timer_conf** –Pointer of LEDC timer configure struct

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Can not find a proper pre-divider number base on the given frequency and the current duty_resolution.

esp_err_t **ledc_update_duty**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel)

LEDC update channel parameters.

Note: Call this function to activate the LEDC updated parameters. After ledc_set_duty, we need to call this function to update the settings. And the new LEDC parameters don't take effect until the next PWM cycle.

Note: ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update

---

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **ledc_set_pin**(int gpio_num, *ledc_mode_t* speed_mode, *ledc_channel_t* ledc_channel)

Set LEDC output gpio.

Note: This function only routes the LEDC signal to GPIO through matrix, other LEDC resources initialization are not involved. Please use `ledc_channel_config()` instead to fully configure a LEDC channel.

Parameters

- **gpio_num** –The LEDC output gpio
- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **ledc_channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **ledc_stop**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t idle_level)

LEDC stop. Disable LEDC output, and set idle level.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **idle_level** –Set output idle level after LEDC stops.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **ledc_set_freq**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_num, uint32_t freq_hz)

LEDC set channel frequency (Hz)

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_num** –LEDC timer index (0-3), select from ledc_timer_t
- **freq_hz** –Set the LEDC frequency

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Can not find a proper pre-divider number base on the given frequency and the current duty_resolution.

uint32_t **ledc_get_freq**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_num)

LEDC get channel frequency (Hz)

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_num** –LEDC timer index (0-3), select from ledc_timer_t

Returns

- 0 error
- Others Current LEDC frequency

esp_err_t **ledc_set_duty_with_hpoint**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t duty, uint32_t hpoint)

LEDC set duty and hpoint value Only after calling ledc_update_duty will the duty update.

Note: ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** –Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- **hpoint** –Set the LEDC hpoint value, the range is [0, (2**duty_resolution)-1]

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

int **ledc_get_hpoint**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel)

LEDC get hpoint value, the counter value when the output is set high level.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns

- LEDC_ERR_VAL if parameter error
- Others Current hpoint value of LEDC channel

esp_err_t **ledc_set_duty**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t duty)

LEDC set duty This function do not change the hpoint value of this channel. if needed, please call ledc_set_duty_with_hpoint. only after calling ledc_update_duty will the duty update.

Note: ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** –Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

uint32_t **ledc_get_duty**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel)

LEDC get duty This function returns the duty at the present PWM cycle. You shouldn't expect the function to return the new duty in the same cycle of calling ledc_update_duty, because duty update doesn't take effect until the next cycle.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns

- LEDC_ERR_DUTY if parameter error
- Others Current LEDC duty

esp_err_t **ledc_set_fade**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t duty, *ledc_duty_direction_t* fade_direction, uint32_t step_num, uint32_t duty_cycle_num, uint32_t duty_scale)

LEDC set gradient Set LEDC gradient, After the function calls the ledc_update_duty function, the function can take effect.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** –Set the start of the gradient duty, the range of duty setting is [0, (2**duty_resolution)]
- **fade_direction** –Set the direction of the gradient
- **step_num** –Set the number of the gradient
- **duty_cycle_num** –Set how many LEDC tick each time the gradient lasts
- **duty_scale** –Set gradient change amplitude

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **ledc_isr_register**(void (*fn)(void*), void *arg, int intr_alloc_flags, *ledc_isr_handle_t* *handle)

Register LEDC interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

Parameters

- **fn** –Interrupt handler function.
- **arg** –User-supplied argument passed to the handler function.
- **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- **handle** –Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Failed to find available interrupt source

esp_err_t **ledc_timer_set**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_sel, uint32_t clock_divider, uint32_t duty_resolution, *ledc_clk_src_t* clk_src)

Configure LEDC settings.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** –Timer index (0-3), there are 4 timers in LEDC module
- **clock_divider** –Timer clock divide value, the timer clock is divided from the selected clock source
- **duty_resolution** –Resolution of duty setting in number of bits. The range is [1, SOC_LEDC_TIMER_BIT_WIDTH]
- **clk_src** –Select LEDC source clock.

Returns

- (-1) Parameter error
- Other Current LEDC duty

esp_err_t **ledc_timer_rst**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_sel)

Reset LEDC timer.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** –LEDC timer index (0-3), select from ledc_timer_t

Returns

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

esp_err_t **ledc_timer_pause**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_sel)

Pause LEDC timer counter.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** –LEDC timer index (0-3), select from ledc_timer_t

Returns

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

esp_err_t **ledc_timer_resume**(*ledc_mode_t* speed_mode, *ledc_timer_t* timer_sel)

Resume LEDC timer.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** –LEDC timer index (0-3), select from ledc_timer_t

Returns

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

esp_err_t **ledc_bind_channel_timer**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, *ledc_timer_t* timer_sel)

Bind LEDC channel with the selected timer.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **timer_sel** –LEDC timer index (0-3), select from ledc_timer_t

Returns

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

esp_err_t **ledc_set_fade_with_step**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, uint32_t scale, uint32_t cycle_num)

Set LEDC fade function.

Note: Call ledc_fade_func_install() once before calling this function. Call ledc_fade_start() after this to start fading.

Note: ledc_set_fade_with_step, ledc_set_fade_with_time and ledc_fade_start are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_fade_step_and_start

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **`speed_mode`** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **`channel`** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **`target_duty`** –Target duty of fading [0, (2**duty_resolution)]
- **`scale`** –Controls the increase or decrease step scale.
- **`cycle_num`** –increase or decrease the duty every cycle_num cycles

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

esp_err_t **`ledc_set_fade_with_time`**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, int max_fade_time_ms)

Set LEDC fade function, with a limited time.

Note: Call ledc_fade_func_install() once before calling this function. Call ledc_fade_start() after this to start fading.

Note: ledc_set_fade_with_step, ledc_set_fade_with_time and ledc_fade_start are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_fade_step_and_start

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **`speed_mode`** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **`channel`** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **`target_duty`** –Target duty of fading [0, (2**duty_resolution)]
- **`max_fade_time_ms`** –The maximum time of the fading (ms).

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

esp_err_t **ledc_fade_func_install** (int intr_alloc_flags)

Install LEDC fade function. This function will occupy interrupt of LEDC module.

Parameters **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Intr flag error
- ESP_ERR_NOT_FOUND Failed to find available interrupt source
- ESP_ERR_INVALID_STATE Fade function already installed

void **ledc_fade_func_uninstall** (void)

Uninstall LEDC fade function.

esp_err_t **ledc_fade_start** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, *ledc_fade_mode_t* fade_mode)

Start LEDC fading.

Note: Call ledc_fade_func_install() once before calling this function. Call this API right after ledc_set_fade_with_time or ledc_set_fade_with_step before to start fading.

Note: Starting fade operation with this API is not thread-safe, use with care.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel number
- **fade_mode** –Whether to block until fading done. See ledc_types.h ledc_fade_mode_t for more info. Note that this function will not return until fading to the target duty if LEDC_FADE_WAIT_DONE mode is selected.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE Channel not initialized or fade function not installed.
- ESP_ERR_INVALID_ARG Parameter error.

esp_err_t **ledc_set_duty_and_update** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t duty, uint32_t hpoint)

A thread-safe API to set duty for LEDC channel and return when duty updated.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.

- **channel** –LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** –Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- **hpoint** –Set the LEDC hpoint value, the range is [0, (2**duty_resolution)-1]

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Fade function init error

esp_err_t **ledc_set_fade_time_and_start** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, uint32_t max_fade_time_ms, *ledc_fade_mode_t* fade_mode)

A thread-safe API to set and start LEDC fade function, with a limited time.

Note: Call ledc_fade_func_install() once, before calling this function.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **target_duty** –Target duty of fading [0, (2**duty_resolution)]
- **max_fade_time_ms** –The maximum time of the fading (ms).
- **fade_mode** –choose blocking or non-blocking mode

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

esp_err_t **ledc_set_fade_step_and_start** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, uint32_t scale, uint32_t cycle_num, *ledc_fade_mode_t* fade_mode)

A thread-safe API to set and start LEDC fade function.

Note: Call ledc_fade_func_install() once before calling this function.

Note: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **target_duty** –Target duty of fading [0, (2**duty_resolution)]
- **scale** –Controls the increase or decrease step scale.

- **cycle_num** –increase or decrease the duty every cycle_num cycles
- **fade_mode** –choose blocking or non-blocking mode

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

esp_err_t **ledc_cb_register** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, *ledc_cbs_t* *cbs, void *user_arg)

LEDC callback registration function.

Note: The callback is called from an ISR, it must never attempt to block, and any FreeRTOS API called must be ISR capable.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** –LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **cbs** –Group of LEDC callback functions
- **user_arg** –user registered data for the callback function

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

Structures

struct **ledc_channel_config_t**

Configuration parameters of LEDC channel for ledc_channel_config function.

Public Members

int **gpio_num**

the LEDC output gpio_num, if you want to use gpio16, gpio_num = 16

ledc_mode_t **speed_mode**

LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode

ledc_channel_t **channel**

LEDC channel (0 - 7)

ledc_intr_type_t **intr_type**

configure interrupt, Fade interrupt enable or Fade interrupt disable

ledc_timer_t **timer_sel**

Select the timer source of channel (0 - 3)

uint32_t **duty**

LEDC channel duty, the range of duty setting is [0, (2**duty_resolution)]

int **hpoint**

LEDC channel hpoint value, the range is [0, (2**duty_resolution)-1]

unsigned int **output_invert**

Enable (1) or disable (0) gpio output invert

struct *ledc_channel_config_t*::[anonymous] **flags**

LEDC flags

struct **ledc_timer_config_t**

Configuration parameters of LEDC Timer timer for ledc_timer_config function.

Public Members

ledc_mode_t **speed_mode**

LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode

ledc_timer_bit_t **duty_resolution**

LEDC channel duty resolution

ledc_timer_t **timer_num**

The timer source of channel (0 - 3)

uint32_t **freq_hz**

LEDC timer frequency (Hz)

ledc_clk_cfg_t **clk_cfg**

Configure LEDC source clock from ledc_clk_cfg_t. Note that LEDC_USE_RTC8M_CLK and LEDC_USE_XTAL_CLK are non-timer-specific clock sources. You can not have one LEDC timer uses RTC8M_CLK as the clock source and have another LEDC timer uses XTAL_CLK as its clock source. All chips except esp32 and esp32s2 do not have timer-specific clock sources, which means clock source for all timers must be the same one.

struct **ledc_cb_param_t**

LEDC callback parameter.

Public Members

ledc_cb_event_t **event**

Event name

uint32_t **speed_mode**

Speed mode of the LEDC channel group

uint32_t **channel**

LEDC channel (0 - LEDC_CHANNEL_MAX-1)

uint32_t **duty**

LEDC current duty of the channel, the range of duty is [0, (2**duty_resolution)]

struct **ledc_cbs_t**

Group of supported LEDC callbacks.

Note: The callbacks are all running under ISR environment

Public Members

ledc_cb_t **fade_cb**

LEDC fade_end callback function

Macros

LEDC_APB_CLK_HZ

Frequency of one of the LEDC peripheral clock sources, APB_CLK.

Note: This macro should have no use in your application, we keep it here only for backward compatible

LEDC_REF_CLK_HZ

Frequency of one of the LEDC peripheral clock sources, REF_TICK.

Note: This macro should have no use in your application, we keep it here only for backward compatible

LEDC_ERR_DUTY

LEDC_ERR_VAL

Type Definitions

typedef *intr_handle_t* **ledc_isr_handle_t**

typedef bool (***ledc_cb_t**)(const *ledc_cb_param_t* *param, void *user_arg)

Type of LEDC event callback.

Param param LEDC callback parameter
Param user_arg User registered data
Return Whether a high priority task has been waken up by this function

Enumerations

enum **ledc_cb_event_t**

LEDC callback event type.

Values:

enumerator **LEDC_FADE_END_EVT**

LEDC fade end event

Header File

- components/hal/include/hal/ledc_types.h

Enumerations

enum **ledc_mode_t**

Values:

enumerator **LEDC_HIGH_SPEED_MODE**

LEDC high speed speed_mode

enumerator **LEDC_LOW_SPEED_MODE**

LEDC low speed speed_mode

enumerator **LEDC_SPEED_MODE_MAX**

LEDC speed limit

enum **ledc_intr_type_t**

Values:

enumerator **LEDC_INTR_DISABLE**

Disable LEDC interrupt

enumerator **LEDC_INTR_FADE_END**

Enable LEDC interrupt

enumerator **LEDC_INTR_MAX**

enum **ledc_duty_direction_t**

Values:

enumerator **LEDC_DUTY_DIR_DECREASE**

LEDC duty decrease direction

enumerator **LEDC_DUTY_DIR_INCREASE**

LEDC duty increase direction

enumerator **LEDC_DUTY_DIR_MAX**

enum **ledc_slow_clk_sel_t**

Values:

enumerator **LEDC_SLOW_CLK_RTC8M**

LEDC low speed timer clock source is 8MHz RTC clock

enumerator **LEDC_SLOW_CLK_APB**

LEDC low speed timer clock source is 80MHz APB clock

enum **ledc_clk_cfg_t**

In theory, the following enumeration shall be placed in LEDC driver's header. However, as the next enumeration, `ledc_clk_src_t`, makes the use of some of these values and to avoid mutual inclusion of the headers, we must define it here.

Values:

enumerator **LEDC_AUTO_CLK**

The driver will automatically select the source clock based on the giving resolution and duty parameter when init the timer

enumerator **LEDC_USE_APB_CLK**

LEDC timer select APB clock as source clock

enumerator **LEDC_USE_RTC8M_CLK**

LEDC timer select RTC8M_CLK as source clock. Only for low speed channels and this parameter must be the same for all low speed channels

enumerator **LEDC_USE_REF_TICK**

LEDC timer select REF_TICK clock as source clock

enum **ledc_clk_src_t**

Values:

enumerator **LEDC_REF_TICK**

LEDC timer clock divided from reference tick (1Mhz)

enumerator **LEDC_APB_CLK**

LEDC timer clock divided from APB clock (80Mhz)

enumerator **LEDC_SCLK**

Selecting this value for LEDC_TICK_SEL_TIMER let the hardware take its source clock from LEDC_APB_CLK_SEL

enum **ledc_timer_t**

Values:

enumerator **LEDC_TIMER_0**

LEDC timer 0

enumerator **LEDC_TIMER_1**

LEDC timer 1

enumerator **LEDC_TIMER_2**

LEDC timer 2

enumerator **LEDC_TIMER_3**

LEDC timer 3

enumerator **LEDC_TIMER_MAX**

enum **ledc_channel_t**

Values:

enumerator **LEDC_CHANNEL_0**

LEDC channel 0

enumerator **LEDC_CHANNEL_1**

LEDC channel 1

enumerator **LEDC_CHANNEL_2**

LEDC channel 2

enumerator **LEDC_CHANNEL_3**

LEDC channel 3

enumerator **LEDC_CHANNEL_4**

LEDC channel 4

enumerator **LEDC_CHANNEL_5**

LEDC channel 5

enumerator **LEDC_CHANNEL_6**

LEDC channel 6

enumerator **LEDC_CHANNEL_7**

LEDC channel 7

enumerator **LEDC_CHANNEL_MAX**

enum **ledc_timer_bit_t**

Values:

enumerator **LEDC_TIMER_1_BIT**

LEDC PWM duty resolution of 1 bits

enumerator **LEDC_TIMER_2_BIT**

LEDC PWM duty resolution of 2 bits

enumerator **LEDC_TIMER_3_BIT**

LEDC PWM duty resolution of 3 bits

enumerator **LEDC_TIMER_4_BIT**

LEDC PWM duty resolution of 4 bits

enumerator **LEDC_TIMER_5_BIT**

LEDC PWM duty resolution of 5 bits

enumerator **LEDC_TIMER_6_BIT**

LEDC PWM duty resolution of 6 bits

enumerator **LEDC_TIMER_7_BIT**

LEDC PWM duty resolution of 7 bits

enumerator **LEDC_TIMER_8_BIT**

LEDC PWM duty resolution of 8 bits

enumerator **LEDC_TIMER_9_BIT**

LEDC PWM duty resolution of 9 bits

enumerator **LEDC_TIMER_10_BIT**

LEDC PWM duty resolution of 10 bits

enumerator **LEDC_TIMER_11_BIT**

LEDC PWM duty resolution of 11 bits

enumerator **LEDC_TIMER_12_BIT**

LEDC PWM duty resolution of 12 bits

enumerator **LEDC_TIMER_13_BIT**

LEDC PWM duty resolution of 13 bits

enumerator **LEDC_TIMER_14_BIT**

LEDC PWM duty resolution of 14 bits

enumerator **LEDC_TIMER_15_BIT**

LEDC PWM duty resolution of 15 bits

enumerator **LEDC_TIMER_16_BIT**

LEDC PWM duty resolution of 16 bits

enumerator **LEDC_TIMER_17_BIT**

LEDC PWM duty resolution of 17 bits

enumerator **LEDC_TIMER_18_BIT**

LEDC PWM duty resolution of 18 bits

enumerator **LEDC_TIMER_19_BIT**

LEDC PWM duty resolution of 19 bits

enumerator **LEDC_TIMER_20_BIT**

LEDC PWM duty resolution of 20 bits

enumerator **LEDC_TIMER_BIT_MAX**

enum **ledc_fade_mode_t**

Values:

enumerator **LEDC_FADE_NO_WAIT**

LEDC fade function will return immediately

enumerator **LEDC_FADE_WAIT_DONE**

LEDC fade function will block until fading to the target duty

enumerator **LEDC_FADE_MAX**

2.6.12 Motor Control Pulse Width Modulator (MCPWM)

The MCPWM peripheral is a versatile PWM generator, which contains various submodules to make it a key element in power electronic applications like motor control, digital power and so on. Typically, the MCPWM peripheral can be used in the following scenarios:

- Digital motor control, e.g. brushed/brushless DC motor, RC servo motor
- Switch mode based digital power conversion
- Power DAC, where the duty cycle is equivalent to a DAC analog value
- Calculate external pulse width, and convert it into other analog value like speed, distance
- Generate Space Vector PWM (SVPWM) signals for Field Oriented Control (FOC)

The main submodules are listed in the following diagram:

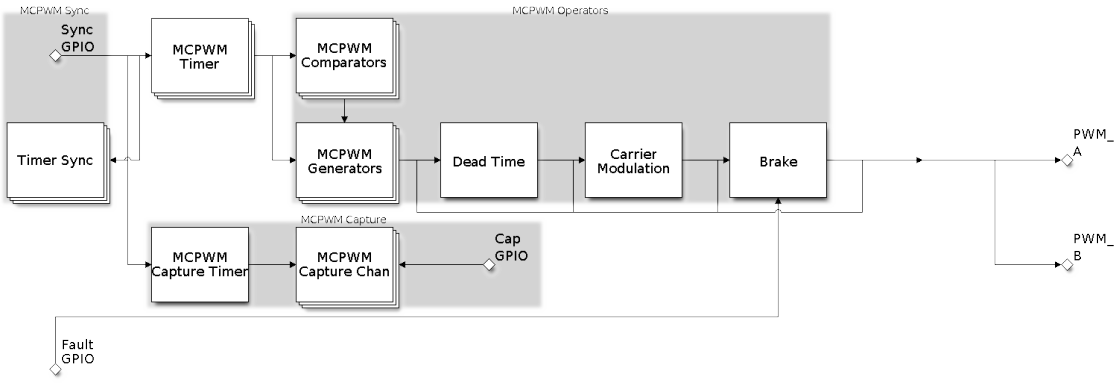


Fig. 12: MCPWM Overview

- **MCPWM Timer**: The time base of the final PWM signal. It also determines the event timing of other submodules.
- **MCPWM Operator**: The key module that is responsible for generating the PWM waveforms. It consists of other submodules, like comparator, PWM generator, dead time, and carrier modulator.
- **MCPWM Comparator**: The compare module takes the time-base count value as input, and continuously compares it to the threshold value configured. When the timer is equal to any of the threshold values, a compare event will be generated and the MCPWM generator can update its level accordingly.

- **MCPWM Generator**: One MCPWM generator can generate a pair of PWM waves, complementarily or independently, based on various events triggered by other submodules like MCPWM Timer and MCPWM Comparator.
- **MCPWM Fault**: The fault module is used to detect the fault condition from outside, mainly via the GPIO matrix. Once the fault signal is active, MCPWM Operator will force all the generators into a predefined state to protect the system from damage.
- **MCPWM Sync**: The sync module is used to synchronize the MCPWM timers, so that the final PWM signals generated by different MCPWM generators can have a fixed phase difference. The sync signal can be routed from the GPIO matrix or from an MCPWM Timer event.
- **Dead Time**: This submodule is used to insert extra delay to the existing PWM edges generated in the previous steps.
- **Carrier Modulation**: The carrier submodule can modulate a high-frequency carrier signal into PWM waveforms by the generator and dead time submodules. This capability is mandatory for controlling the power-switching elements.
- **Brake**: MCPWM operator can set how to brake the generators when a particular fault is detected. You can shut down the PWM output immediately or regulate the PWM output cycle by cycle, depending on how critical the fault is.
- **MCPWM Capture**: This is a standalone submodule that can work even without the above MCPWM operators. The capture consists one dedicated timer and several independent channels, with each channel connected to the GPIO. A pulse on the GPIO triggers the capture timer to store the time-base count value and then notify you by an interrupt. Using this feature, you can measure a pulse width precisely. What is more, the capture timer can also be synchronized by the MCPWM Sync submodule.

Functional Overview

Description of the MCPWM functionality is divided into the following sections:

- *Resource Allocation and Initialization* - covers how to allocate various MCPWM objects, like timers, operators, comparators, generators and so on. These objects are the basis of the following IO setting and control functions.
- *Timer Operations and Events* - describes control functions and event callbacks that supported by the MCPWM timer.
- *Comparator Operations and Events* - describes control functions and event callbacks that supported by the MCPWM comparator.
- *Generator Actions on Events* - describes how to set actions for MCPWM generators on particular events that generated by the MCPWM timer and comparators.
- *Classical PWM Waveforms and Generator Configurations* - demonstrates some classical PWM waveforms that can be achieved by configuring generator actions.
- *Dead Time* - describes how to set dead time for MCPWM generators.
- *Classical PWM Waveforms and Dead Time Configurations* - demonstrates some classical PWM waveforms that can be achieved by configuring dead time.
- *Carrier Modulation* - describes how to set modulate a high frequency onto the final PWM waveforms.
- *Faults and Brake Actions* - describes how to set brake actions for MCPWM operators on particular fault event.
- *Generator Force Actions* - describes how to control the generator output level asynchronously in a forceful way.
- *Synchronization* - describes how to synchronize the MCPWM timers and get a fixed phase difference between the generated PWM signals.
- *Capture* - describes how to use the MCPWM capture module to measure the pulse width of a signal.
- *Power Management* - describes how different source clock will affect power consumption.
- *IRAM Safe* - describes tips on how to make the RMT interrupt work better along with a disabled cache.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.
- *Kconfig Options* - lists the supported Kconfig options that can bring different effects to the driver.

Resource Allocation and Initialization As displayed in the diagram above, the MCPWM peripheral consists of several submodules. Each submodule has its own resource allocation, which is described in the following sections.

MCPWM Timers You can allocate a MCPWM timer object by calling `mcpwm_new_timer()` function, with a configuration structure `mcpwm_timer_config_t` as the parameter. The configuration structure is defined as:

- *mcpwm_timer_config_t::group_id* specifies the MCPWM group ID. The ID should belong to [0, *SOC_MCPWM_GROUPS* - 1] range. Please note, timers located in different groups are totally independent.
- *mcpwm_timer_config_t::intr_priority* sets the priority of the interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.
- *mcpwm_timer_config_t::clk_src* sets the clock source of the timer.
- *mcpwm_timer_config_t::resolution_hz* set the expected resolution of the timer, the driver internally will set a proper divider based on the clock source and the resolution.
- *mcpwm_timer_config_t::count_mode* sets the count mode of the timer.
- *mcpwm_timer_config_t::period_ticks* sets the period of the timer, in ticks (the tick resolution is set in the *mcpwm_timer_config_t::resolution_hz*).
- *mcpwm_timer_config_t::update_period_on_empty* sets whether to update the period value when the timer counts to zero.
- *mcpwm_timer_config_t::update_period_on_sync* sets whether to update the period value when the timer takes a sync signal.

The *mcpwm_new_timer()* will return a pointer to the allocated timer object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free timers in the MCPWM group, this function will return *ESP_ERR_NOT_FOUND* error.[1]

On the contrary, calling *mcpwm_del_timer()* function will free the allocated timer object.

MCPWM Operators You can allocate a MCPWM operator object by calling *mcpwm_new_operator()()* function, with a configuration structure *mcpwm_operator_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_operator_config_t::group_id* specifies the MCPWM group ID. The ID should belong to [0, *SOC_MCPWM_GROUPS* - 1] range. Please note, operators located in different groups are totally independent.
- *mcpwm_operator_config_t::intr_priority* sets the priority of the interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.
- *mcpwm_operator_config_t::update_gen_action_on_tez* sets whether to update the generator action when the timer counts to zero. Here and below, the timer refers to the one that is connected to the operator by *mcpwm_operator_connect_timer()*.
- *mcpwm_operator_config_t::update_gen_action_on_tep* sets whether to update the generator action when the timer counts to peak.
- *mcpwm_operator_config_t::update_gen_action_on_sync* sets whether to update the generator action when the timer takes a sync signal.
- *mcpwm_operator_config_t::update_dead_time_on_tez* sets whether to update the dead time when the timer counts to zero.
- *mcpwm_operator_config_t::update_dead_time_on_tep* sets whether to update the dead time when the timer counts to peak.
- *mcpwm_operator_config_t::update_dead_time_on_sync* sets whether to update the dead time when the timer takes a sync signal.

The *mcpwm_new_operator()()* will return a pointer to the allocated operator object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free operators in the MCPWM group, this function will return *ESP_ERR_NOT_FOUND* error.[1]

On the contrary, calling *mcpwm_del_operator()()* function will free the allocated operator object.

MCPWM Comparators You can allocate a MCPWM comparator object by calling *mcpwm_new_comparator()* function, with a MCPWM operator handle and configuration structure *mcpwm_comparator_config_t* as the parameter. The operator handle is created by *mcpwm_new_operator()()*. The configuration structure is defined as:

- *mcpwm_comparator_config_t::intr_priority* sets the priority of the interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.

[1] Different ESP chip series might have different number of MCPWM resources (e.g. groups, timers, comparators, operators, generators and so on). Please refer to the [TRM] for details. The driver won't forbid you from applying for more MCPWM resources, but it will return error when there's no hardware resources available. Please always check the return value when doing *Resource Allocation*.

- *mcpwm_comparator_config_t::update_cmp_on_tez* sets whether to update the compare threshold when the timer counts to zero.
- *mcpwm_comparator_config_t::update_cmp_on_tep* sets whether to update the compare threshold when the timer counts to peak.
- *mcpwm_comparator_config_t::update_cmp_on_sync* sets whether to update the compare threshold when the timer takes a sync signal.

The *mcpwm_new_comparator()* will return a pointer to the allocated comparator object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free comparators in the MCPWM operator, this function will return *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

On the contrary, calling *mcpwm_del_comparator()* function will free the allocated comparator object.

MCPWM Generators You can allocate a MCPWM generator object by calling *mcpwm_new_generator()* function, with a MCPWM operator handle and configuration structure *mcpwm_generator_config_t* as the parameter. The operator handle is created by *mcpwm_new_operator()()*. The configuration structure is defined as:

- *mcpwm_generator_config_t::gen_gpio_num* sets the GPIO number used by the generator.
- *mcpwm_generator_config_t::invert_pwm* sets whether to invert the PWM signal.
- *mcpwm_generator_config_t::io_loop_back* sets whether to enable the Loop-back mode. It is for debugging purposes only. It enables both the GPIO's input and output ability through the GPIO matrix peripheral.
- *mcpwm_generator_config_t::io_od_mode* configures the PWM GPIO as open-drain output.
- *mcpwm_generator_config_t::pull_up* and *mcpwm_generator_config_t::pull_down* controls whether to enable the internal pull-up and pull-down resistors accordingly.

The *mcpwm_new_generator()* will return a pointer to the allocated generator object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free generators in the MCPWM operator, this function will return *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

On the contrary, calling *mcpwm_del_generator()* function will free the allocated generator object.

MCPWM Faults There are two types of faults: A fault signal reflected from the GPIO and a fault generated by software. To allocate a GPIO fault object, you can call *mcpwm_new_gpio_fault()* function, with configuration structure *mcpwm_gpio_fault_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_gpio_fault_config_t::group_id* sets the MCPWM group ID. The ID should belong to [0, *SOC_MCPWM_GROUPS* - 1] range. Please note, GPIO fault located in different groups are totally independent, i.e. GPIO fault in group 0 can not be detected by the operator in group 1.
- *mcpwm_gpio_fault_config_t::gpio_num* sets the GPIO number used by the fault.
- *mcpwm_gpio_fault_config_t::active_level* sets the active level of the fault signal.
- *mcpwm_gpio_fault_config_t::pull_up* and *mcpwm_gpio_fault_config_t::pull_down* set whether to pull up and/or pull down the GPIO internally.
- *mcpwm_gpio_fault_config_t::io_loop_back* sets whether to enable the loop back mode. It is for debugging purposes only. It enables both the GPIO's input and output ability through the GPIO matrix peripheral.

The *mcpwm_new_gpio_fault()* will return a pointer to the allocated fault object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free GPIO faults in the MCPWM group, this function will return *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

Software fault object can be used to trigger a fault by calling a function *mcpwm_soft_fault_activate()* instead of waiting for a real fault signal on the GPIO. A software fault object can be allocated by calling *mcpwm_new_soft_fault()* function, with configuration structure *mcpwm_soft_fault_config_t* as the parameter. Currently this configuration structure is left for future purpose. *mcpwm_new_soft_fault()* function will return a pointer to the allocated fault object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no memory left for the fault object, this function will return *ESP_ERR_NO_MEM* error. Although the software fault and GPIO fault are of different types, but the returned fault handle is of the same type.

On the contrary, calling *mcpwm_del_fault()* function will free the allocated fault object, this function works for both software and GPIO fault.

MCPWM Sync Sources The sync source is what can be used to synchronize the MCPWM timer and MCPWM capture timer. There're three types of sync sources: A sync source reflected from the GPIO, a sync source generated by software and a sync source generated by MCPWM timer event.

To allocate a GPIO sync source, you can call *mcpwm_new_gpio_sync_src()* function, with configuration structure *mcpwm_gpio_sync_src_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_gpio_sync_src_config_t::group_id* sets the MCPWM group ID. The ID should belong to [0, *SOC_MCPWM_GROUPS* - 1] range. Please note, GPIO sync source located in different groups are totally independent, i.e. GPIO sync source in group 0 can not be detected by the timers in group 1.
- *mcpwm_gpio_sync_src_config_t::gpio_num* sets the GPIO number used by the sync source.
- *mcpwm_gpio_sync_src_config_t::active_neg* sets whether the sync signal is active on falling edges.
- *mcpwm_gpio_sync_src_config_t::pull_up* and *mcpwm_gpio_sync_src_config_t::pull_down* set whether to pull up and/or pull down the GPIO internally.
- *mcpwm_gpio_sync_src_config_t::io_loop_back* sets whether to enable the loop back mode. It is for debugging purposes only. It enables both the GPIO's input and output ability through the GPIO matrix peripheral.

The *mcpwm_new_gpio_sync_src()* will return a pointer to the allocated sync source object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no more free GPIO sync sources in the MCPWM group, this function will return *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

To allocate a Timer event sync source, you can call *mcpwm_new_timer_sync_src()* function, with configuration structure *mcpwm_timer_sync_src_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_timer_sync_src_config_t::timer_event* specifies on what timer event to generate the sync signal.
- *mcpwm_timer_sync_src_config_t::propagate_input_sync* sets whether to propagate the input sync signal (i.e. the input sync signal will be routed to its sync output).

The *mcpwm_new_timer_sync_src()* will return a pointer to the allocated sync source object if the allocation succeeds. Otherwise, it will return error code. Specifically, if a sync source has been allocated from the same timer before, this function will return *ESP_ERR_INVALID_STATE* error.

Last but not least, to allocate a software sync source, you can call *mcpwm_new_soft_sync_src()* function, with configuration structure *mcpwm_soft_sync_config_t* as the parameter. Currently this configuration structure is left for future purpose. *mcpwm_new_soft_sync_src()* will return a pointer to the allocated sync source object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no memory left for the sync source object, this function will return *ESP_ERR_NO_MEM* error. Please note, to make a software sync source take effect, don't forget to call *mcpwm_soft_sync_activate()*.

On the contrary, calling *mcpwm_del_sync_src()* function will free the allocated sync source object, this function works for all types of sync sources.

MCPWM Capture Timer and Channels The MCPWM group has a dedicated timer which is used to capture the timestamp when specific event occurred. The capture timer is connected with several independent channels, each channel is assigned with a GPIO.

To allocate a capture timer, you can call *mcpwm_new_capture_timer()* function, with configuration structure *mcpwm_capture_timer_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_capture_timer_config_t::group_id* sets the MCPWM group ID. The ID should belong to [0, *SOC_MCPWM_GROUPS* - 1] range.
- *mcpwm_capture_timer_config_t::clk_src* sets the clock source of the capture timer.

The *mcpwm_new_capture_timer()* will return a pointer to the allocated capture timer object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no free capture timer left in the MCPWM group, this function will return *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

Next, to allocate a capture channel, you can call *mcpwm_new_capture_channel()* function, with a capture timer handle and configuration structure *mcpwm_capture_channel_config_t* as the parameter. The configuration structure is defined as:

- *mcpwm_capture_channel_config_t::intr_priority* sets the priority of the interrupt. If it is set to `0`, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.
- *mcpwm_capture_channel_config_t::gpio_num* sets the GPIO number used by the capture channel.
- *mcpwm_capture_channel_config_t::prescale* sets the prescaler of the input signal.
- *mcpwm_capture_channel_config_t::pos_edge* and *mcpwm_capture_channel_config_t::neg_edge* set whether to capture on the positive and/or falling edge of the input signal.
- *mcpwm_capture_channel_config_t::pull_up* and *mcpwm_capture_channel_config_t::pull_down* set whether to pull up and/or pull down the GPIO internally.
- *mcpwm_capture_channel_config_t::invert_cap_signal* sets whether to invert the capture signal.
- *mcpwm_capture_channel_config_t::io_loop_back* sets whether to enable the loop back mode. It is for debugging purposes only. It enables both the GPIO's input and output ability through the GPIO matrix peripheral.

The *mcpwm_new_capture_channel()* will return a pointer to the allocated capture channel object if the allocation succeeds. Otherwise, it will return error code. Specifically, when there are no free capture channel left in the capture timer, this function will return *ESP_ERR_NOT_FOUND* error.

On the contrary, calling *mcpwm_del_capture_channel()* and *mcpwm_del_capture_timer()* will free the allocated capture channel and timer object accordingly.

MCPWM interrupt priority MCPWM allows configuring interrupts separately for timer, operator, comparator, fault, and capture events. The interrupt priority is determined by the respective `config_t::intr_priority`. Additionally, events within the same MCPWM group share a common interrupt source. When registering multiple interrupt events, the interrupt priorities need to remain consistent.

Note: When registering multiple interrupt events within an MCPWM group, the driver will use the interrupt priority of the first registered event as the MCPWM group's interrupt priority.

Timer Operations and Events

Update Period The timer period is initialized by the *mcpwm_timer_config_t::period_ticks* parameter in *mcpwm_timer_config_t*. You can update the period at runtime by calling *mcpwm_timer_set_period()* function. The new period will take effect based on how you set the *mcpwm_timer_config_t::update_period_on_empty* and *mcpwm_timer_config_t::update_period_on_sync* parameters in *mcpwm_timer_config_t*. If none of them are set, the timer period will take effect immediately.

Register Timer Event Callbacks The MCPWM timer can generate different events at runtime. If you have some function that should be called when particular event happens, you should hook your function to the interrupt service routine by calling *mcpwm_timer_register_event_callbacks()*. The callback function prototype is declared in *mcpwm_timer_event_cb_t*. All supported event callbacks are listed in the *mcpwm_timer_event_callbacks_t*:

- *mcpwm_timer_event_callbacks_t::on_full* sets callback function for timer when it counts to peak value.

- *mcpwm_timer_event_callbacks_t::on_empty* sets callback function for timer when it counts to zero.
- *mcpwm_timer_event_callbacks_t::on_stop* sets callback function for timer when it is stopped.

The callback functions above are called within the ISR context, so they should **not** attempt to block (e.g., make sure that only FreeRTOS APIs with `ISR` suffix is called within the function).

The parameter `user_data` of *mcpwm_timer_register_event_callbacks()* function is used to save user's own context, it will be passed to each callback function directly.

This function will lazy install interrupt service for the MCPWM timer without enabling it. It is only allowed to be called before before *mcpwm_timer_enable()*, otherwise the *ESP_ERR_INVALID_STATE* error will be returned. See also *Enable and Disable timer* for more information.

Enable and Disable Timer Before doing IO control to the timer, user needs to enable the timer first, by calling *mcpwm_timer_enable()*. Internally, this function will:

- switch the timer state from **init** to **enable**.
- enable the interrupt service if it has been lazy installed by *mcpwm_timer_register_event_callbacks()*.
- acquire a proper power management lock if a specific clock source (e.g. PLL_160M clock) is selected. See also *Power management* for more information.

On the contrary, calling *mcpwm_timer_disable()* will put the timer driver back to **init** state, disable the interrupts service and release the power management lock.

Start and Stop Timer The basic IO operation of a timer is to start and stop. Calling *mcpwm_timer_start_stop()* with different *mcpwm_timer_start_stop_cmd_t* commands can start the timer immediately or stop the timer at a specific event. What's more, you can even start the timer for only one round, that means, the timer will count to peak value or zero, and then stop itself.

Connect Timer with Operator The allocated MCPWM Timer should be connected with a MCPWM operator by calling *mcpwm_operator_connect_timer()*, so that the operator can take that timer as its time base, and generate the required PWM waves. Make sure the MCPWM timer and operator are in the same group, otherwise, this function will return *ESP_ERR_INVALID_ARG* error.

Comparator Operations and Events

Register Event Callbacks The MCPWM comparator can inform the user when the timer counter equals to the compare value. If you have some function that should be called when this event happens, you should hook your function to the interrupt service routine by calling *mcpwm_comparator_register_event_callbacks()*. The callback function prototype is declared in *mcpwm_compare_event_cb_t*. All supported event callbacks are listed in the *mcpwm_comparator_event_callbacks_t*:

- *mcpwm_comparator_event_callbacks_t::on_reach* sets callback function for comparator when the timer counter equals to the compare value.

The callback function will provide event specific data of type *mcpwm_compare_event_data_t* to the user. The callback function is called within the ISR context, so is should **not** attempt to block (e.g., make sure that only FreeRTOS APIs with `ISR` suffix is called within the function).

The parameter `user_data` of *mcpwm_comparator_register_event_callbacks()* function is used to save user's own context, it will be passed to the callback function directly.

This function will lazy install interrupt service for the MCPWM comparator, whereas the service can only be removed in *mcpwm_del_comparator*.

Set Compare Value You can set the compare value for the MCPWM comparator at runtime by calling *mcpwm_comparator_set_compare_value()*. There's a few points to note:

- New compare value might won't take effect immediately. The update time for the compare value is set by *mcpwm_comparator_config_t::update_cmp_on_tez* or *mcpwm_comparator_config_t::update_cmp_on_tep* or *mcpwm_comparator_config_t::update_cmp*
- Make sure the operator has connected to one MCPWM timer already by *mcpwm_operator_connect_timer()*. Otherwise, it will return error code *ESP_ERR_INVALID_STATE*.
- The compare value shouldn't exceed timer's count peak, otherwise, the compare event will never got triggered.

Generator Actions on Events

Set Generator Action on Timer Event One generator can set multiple actions on different timer events, by calling *mcpwm_generator_set_actions_on_timer_event()* with variable number of action configurations. The action configuration is defined in *mcpwm_gen_timer_event_action_t*:

- *mcpwm_gen_timer_event_action_t::direction* specific the timer direction. The supported directions are listed in *mcpwm_timer_direction_t*.
- *mcpwm_gen_timer_event_action_t::event* specifies the timer event. The supported timer events are listed in *mcpwm_timer_event_t*.
- *mcpwm_gen_timer_event_action_t::action* specifies the generator action to be taken. The supported actions are listed in *mcpwm_generator_action_t*.

There's a helper macro *MCPWM_GEN_TIMER_EVENT_ACTION* to simplify the construction of a timer event action entry.

Please note, the argument list of *mcpwm_generator_set_actions_on_timer_event()* **must** be terminated by *MCPWM_GEN_TIMER_EVENT_ACTION_END*.

You can also set the timer action one by one by calling *mcpwm_generator_set_action_on_timer_event()* without varargs.

Set Generator Action on Compare Event One generator can set multiple actions on different compare events, by calling *mcpwm_generator_set_actions_on_compare_event()* with variable number of action configurations. The action configuration is defined in *mcpwm_gen_compare_event_action_t*:

- *mcpwm_gen_compare_event_action_t::direction* specific the timer direction. The supported directions are listed in *mcpwm_timer_direction_t*.
- *mcpwm_gen_compare_event_action_t::comparator* specifies the comparator handle. See *MCPWM Comparators* for how to allocate a comparator.
- *mcpwm_gen_compare_event_action_t::action* specifies the generator action to be taken. The supported actions are listed in *mcpwm_generator_action_t*.

There's a helper macro *MCPWM_GEN_COMPARE_EVENT_ACTION* to simplify the construction of a compare event action entry.

Please note, the argument list of *mcpwm_generator_set_actions_on_compare_event()* **must** be terminated by *MCPWM_GEN_COMPARE_EVENT_ACTION_END*.

You can also set the compare action one by one by calling *mcpwm_generator_set_action_on_compare_event()* without varargs.

Set Generator Action on Fault Event One generator can set action on fault based trigger events, by calling *mcpwm_generator_set_action_on_fault_event()* with an action configurations. The action configuration is defined in *mcpwm_gen_fault_event_action_t*:

- *mcpwm_gen_fault_event_action_t::direction* specifies the timer direction. The supported directions are listed in *mcpwm_timer_direction_t*.

- *mcpwm_gen_fault_event_action_t::fault* specifies the fault used for the trigger. See *MCPWM Faults* for how to allocate a fault.
- *mcpwm_gen_fault_event_action_t::action* specifies the generator action to be taken. The supported actions are listed in *mcpwm_generator_action_t*.

When no free trigger slot is left in the operator to which the generator belongs, this function will return the *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

The trigger only support GPOI fault. when the input is not a GPIO fault, this function will return the *ESP_ERR_NOT_SUPPORTED* error.

There's a helper macro *MCPWM_GEN_FAULT_EVENT_ACTION* to simplify the construction of a trigger event action entry.

Please note, fault event does not have variadic function like `mcpwm_generator_set_actions_on_fault_event()`.

Set Generator Action on Sync Event One generator can set action on sync based trigger events, by calling *mcpwm_generator_set_action_on_sync_event()* with an action configurations. The action configuration is defined in *mcpwm_gen_sync_event_action_t*:

- *mcpwm_gen_sync_event_action_t::direction* specifies the timer direction. The supported directions are listed in *mcpwm_timer_direction_t*.
- *mcpwm_gen_sync_event_action_t::sync* specifies the sync source used for the trigger. See *MCPWM Sync Sources* for how to allocate a sync source.
- *mcpwm_gen_sync_event_action_t::action* specifies the generator action to be taken. The supported actions are listed in *mcpwm_generator_action_t*.

When no free trigger slot is left in the operator to which the generator belongs, this function will return the *ESP_ERR_NOT_FOUND* error.[Page 1173, 1]

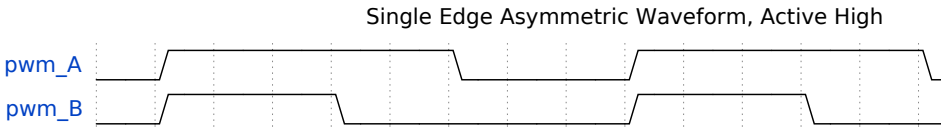
The trigger only support one sync action, regardless of the kinds. When set sync actions more than once, this function will return the *ESP_ERR_INVALID_STATE* error.

There's a helper macro *MCPWM_GEN_SYNC_EVENT_ACTION* to simplify the construction of a trigger event action entry.

Please note, sync event does not have variadic function like `mcpwm_generator_set_actions_on_sync_event()`.

Classical PWM Waveforms and Generator Configurations This section will demonstrate the classical PWM waveforms that can be generated by the pair of the generators. The code snippet that is used to generate the waveforms is also provided below the diagram. Some general summary:

- The **Symmetric** or **Asymmetric** of the waveforms are determined by the count mode of the MCPWM timer.
- The **active level** of the waveform pair is determined by the level of the PWM with a smaller duty cycle.
- The period of the PWM waveform is determined by the timer's period and count mode.
- The duty cycle of the PWM waveform is determined by the generator's various action combinations.



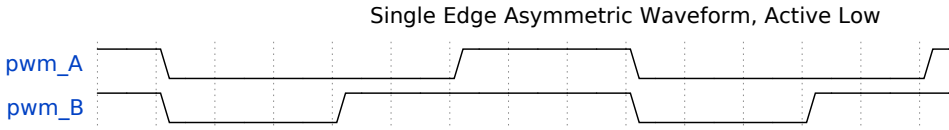
Asymmetric Single Edge Active High

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
```

(continues on next page)

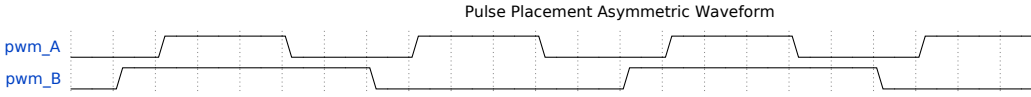
(continued from previous page)

```
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW)));
}
```



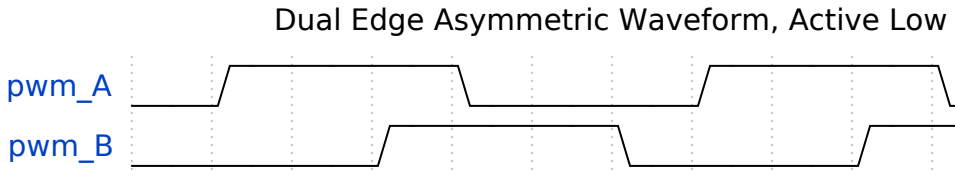
Asymmetric Single Edge Active Low

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_FULL, MCPWM_GEN_ACTION_LOW)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_FULL, MCPWM_GEN_ACTION_LOW)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_HIGH)));
}
```



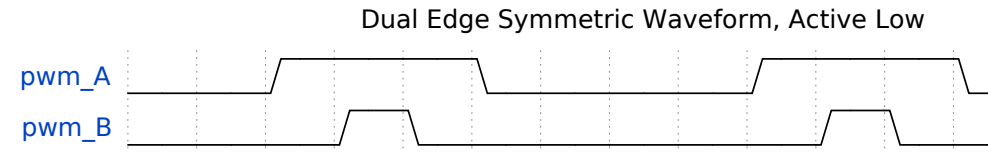
Asymmetric Pulse Placement

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_TOGGLE),
                    MCPWM_GEN_TIMER_EVENT_ACTION_END()));
}
```



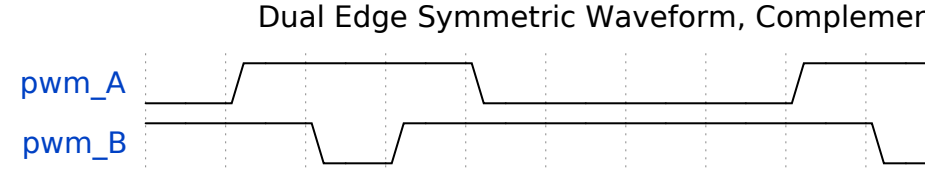
Asymmetric Dual Edge Active Low

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN,␣
↪cmpb, MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN, MCPWM_
↪TIMER_EVENT_FULL, MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_TIMER_EVENT_ACTION_END()));
}
```



Symmetric Dual Edge Active Low

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN,␣
↪cmpa, MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN,␣
↪cmpb, MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
}
```



Symmetric Dual Edge Complementary

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN,␣
↪cmpa, MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
    ESP_ERROR_CHECK(mcpwm_generator_set_actions_on_compare_event(genb,
```

(continues on next page)

(continued from previous page)

```
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW),
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_DOWN,␣
↪cmpb, MCPWM_GEN_ACTION_HIGH),
                    MCPWM_GEN_COMPARE_EVENT_ACTION_END()));
}
```

Dead Time In power electronics, the rectifier and inverter are commonly used. This requires the use of rectifier bridge and inverter bridge. Each bridge arm has two power electronic devices, such as MOSFET, IGBT, etc. The two MOSFETs on the same arm can't conduct at the same time, otherwise there will be a short circuit. The fact is that, although the PWM wave shows it is turning off the switch, but the MOSFET still needs a small time window to make that happen. This requires an extra delay to be added to the existing PWM wave that generated by setting *Generator Actions on Events*.

The dead time driver works like a **decorator**. This is also reflected in the function parameters of *mcpwm_generator_set_dead_time()*, where it takes the primary generator handle (`in_generator`), and returns a new generator (`out_generator`) after applying the dead time. Please note, if the `out_generator` and `in_generator` are the same, it means you are adding the time delay to the PWM waveform in an "in-place" fashion. In turn, if the `out_generator` and `in_generator` are different, it means you are deriving a new PWM waveform from the existing `in_generator`.

Dead-time specific configuration is listed in the *mcpwm_dead_time_config_t* structure:

- *mcpwm_dead_time_config_t::posedge_delay_ticks* and *mcpwm_dead_time_config_t::negedge_de* set the number of ticks to delay the PWM waveform on the rising and falling edge. Specifically, setting both of them to zero means to bypass the dead-time module. The resolution of the dead-time tick is the same to the timer that is connected with the operator by *mcpwm_operator_connect_timer()*.
- *mcpwm_dead_time_config_t::invert_output*: Whether to invert the signal after applying the dead-time, which can be used to control the delay edge polarity.

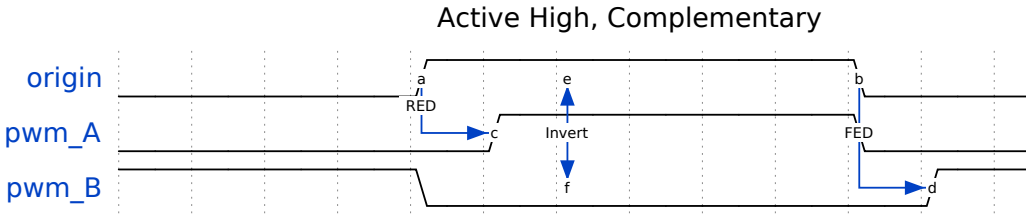
Warning: Due to the hardware limitation, one delay module (either *posedge delay* or *negedge delay*) can't be applied to multiple MCPWM generators at the same time. e.g. the following configuration is **invalid**:

```
mcpwm_dead_time_config_t dt_config = {
    .posedge_delay_ticks = 10,
};
// Set posedge delay to generator A
mcpwm_generator_set_dead_time(mcpwm_gen_a, mcpwm_gen_a, &dt_config);
// NOTE: This is invalid, you can't apply the posedge delay to another generator
mcpwm_generator_set_dead_time(mcpwm_gen_b, mcpwm_gen_b, &dt_config);
```

However, you can apply *posedge delay* to generator A and *negedge delay* to generator B. You can also set both *posedge delay* and *negedge delay* for generator A, while letting generator B bypass the dead time module.

Note: It is also possible to generate the required dead time by setting *Generator Actions on Events*, especially by controlling edge placement using different comparators. However, if the more classical edge delay-based dead time with polarity control is required, then the dead-time submodule should be used.

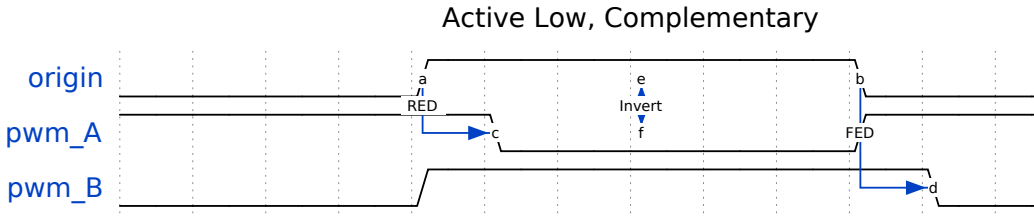
Classical PWM Waveforms and Dead Time Configurations This section will demonstrate the classical PWM waveforms that can be generated by the dead-time submodule. The code snippet that is used to generate the waveforms is also provided below the diagram.



Active High Complementary

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 50,
        .negedge_delay_ticks = 0
    };
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    dead_time_config.posedge_delay_ticks = 0;
    dead_time_config.negedge_delay_ticks = 100;
    dead_time_config.flags.invert_output = true;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, genb, &dead_time_config));
}
```



Active Low Complementary

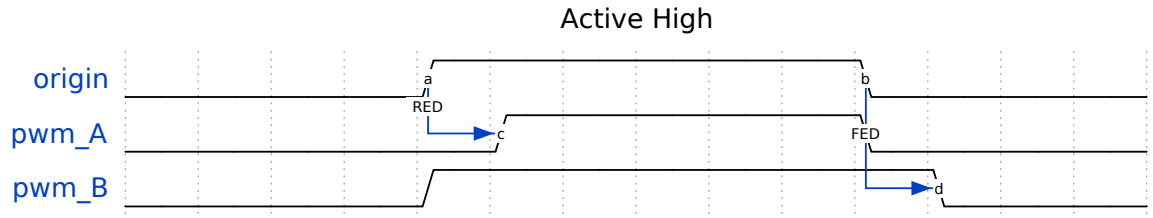
```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 50,
        .negedge_delay_ticks = 0,
        .flags.invert_output = true
    };
```

(continues on next page)

(continued from previous page)

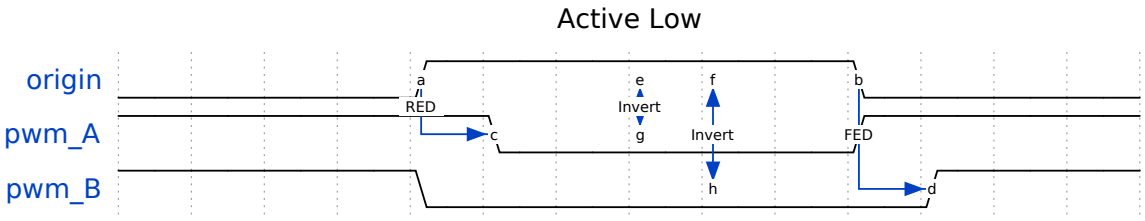
```
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    dead_time_config.posedge_delay_ticks = 0;
    dead_time_config.negedge_delay_ticks = 100;
    dead_time_config.flags.invert_output = false;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, genb, &dead_time_config));
}
```



Active High

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 50,
        .negedge_delay_ticks = 0,
    };
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    dead_time_config.posedge_delay_ticks = 0;
    dead_time_config.negedge_delay_ticks = 100;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, genb, &dead_time_config));
}
```



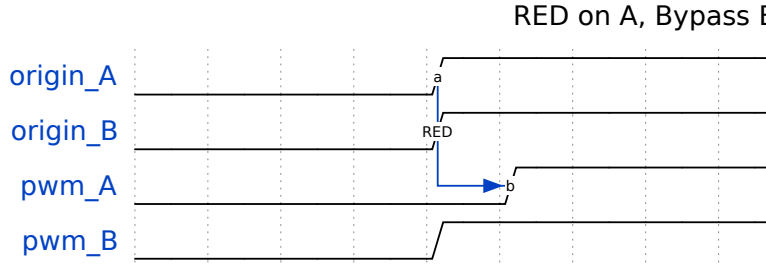
Active Low

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
}
```

(continues on next page)

(continued from previous page)

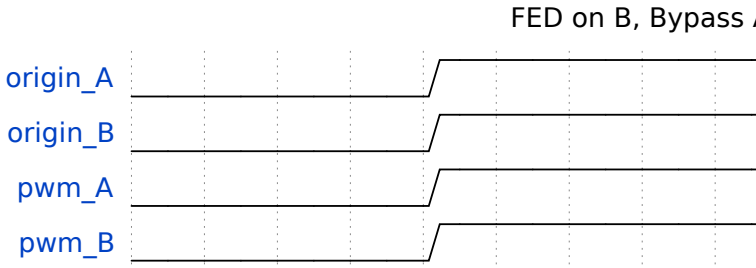
```
static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 50,
        .negedge_delay_ticks = 0,
        .flags.invert_output = true
    };
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    dead_time_config.posedge_delay_ticks = 0;
    dead_time_config.negedge_delay_ticks = 100;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, genb, &dead_time_config));
}
```



Rising Delay on PWMA, Bypass deadtime for PWMB

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 50,
        .negedge_delay_ticks = 0,
    };
    // apply deadtime to generator_a
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    // bypass deadtime module for generator_b
    dead_time_config.posedge_delay_ticks = 0;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(genb, genb, &dead_time_config));
}
```



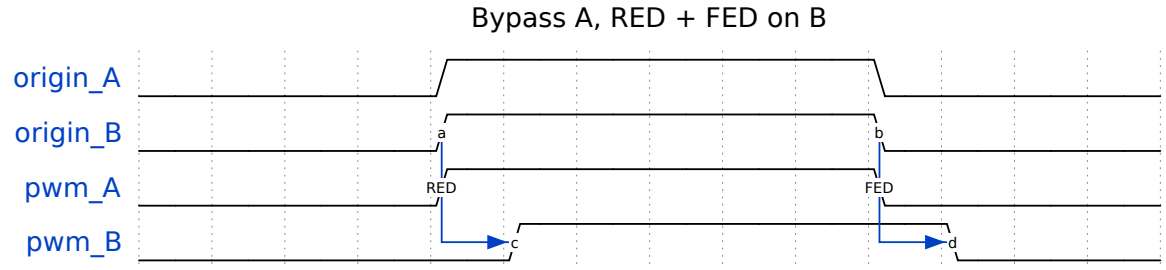
Falling Delay on PWMB, Bypass deadtime for PWMA

```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 0,
        .negedge_delay_ticks = 0,
    };
    // generator_a bypass the deadtime module (no delay)
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    // apply dead time to generator_b
    dead_time_config.negedge_delay_ticks = 50;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(genb, genb, &dead_time_config));

}
```

Rising and Falling Delay on PWMB, Bypass deadtime for PWMA




```
static void gen_action_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb,␣
↪mcpwm_cmpr_handle_t cmpa, mcpwm_cmpr_handle_t cmpb)
{
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(gena,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
```

(continues on next page)

(continued from previous page)

```
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(gena,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpa,␣
↪MCPWM_GEN_ACTION_LOW)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_timer_event(genb,
                    MCPWM_GEN_TIMER_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, MCPWM_
↪TIMER_EVENT_EMPTY, MCPWM_GEN_ACTION_HIGH)));
    ESP_ERROR_CHECK(mcpwm_generator_set_action_on_compare_event(genb,
                    MCPWM_GEN_COMPARE_EVENT_ACTION(MCPWM_TIMER_DIRECTION_UP, cmpb,␣
↪MCPWM_GEN_ACTION_LOW)));
}

static void dead_time_config(mcpwm_gen_handle_t gena, mcpwm_gen_handle_t genb)
{
    mcpwm_dead_time_config_t dead_time_config = {
        .posedge_delay_ticks = 0,
        .negedge_delay_ticks = 0,
    };
    // generator_a bypass the deadtime module (no delay)
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(gena, gena, &dead_time_config));
    // apply dead time on both edge for generator_b
    dead_time_config.negedge_delay_ticks = 50;
    dead_time_config.posedge_delay_ticks = 50;
    ESP_ERROR_CHECK(mcpwm_generator_set_dead_time(genb, genb, &dead_time_config));
}
```

Carrier Modulation The MCPWM operator has a carrier submodule that can be used if galvanic isolation from the motor driver is required (e.g. isolated digital power application) by passing the PWM output signals through transformers. Any of PWM output signals may be at 100% duty and not changing whenever motor is required to run steady at the full load. Coupling of non alternating signals with a transformer is problematic, so the signals are modulated by the carrier submodule to create an AC waveform, to make the coupling possible.

To configure the carrier submodule, you can call *mcpwm_operator_apply_carrier()*, and provide configuration structure *mcpwm_carrier_config_t*:

- *mcpwm_carrier_config_t::clk_src* sets the clock source of the carrier.
- *mcpwm_carrier_config_t::frequency_hz* indicates carrier frequency in Hz.
- *mcpwm_carrier_config_t::duty_cycle* indicates the duty cycle of the carrier. Note that, the supported choices of the duty cycle are discrete, the driver will search for the nearest one based on your configuration.
- *mcpwm_carrier_config_t::first_pulse_duration_us* indicates the duration of the first pulse in microseconds. The resolution of the first pulse duration is determined by the carrier frequency you set in the *mcpwm_carrier_config_t::frequency_hz*. The first pulse duration can't be zero, and it has to be at least one period of the carrier. A longer pulse width can help conduct the inductance quicker.
- *mcpwm_carrier_config_t::invert_before_modulate* and *mcpwm_carrier_config_t::invert_after_modulate* set whether to invert the carrier output before and after modulation.

Specifically, the carrier submodule can be disabled by calling *mcpwm_operator_apply_carrier()* with a `NULL` configuration.

Faults and Brake Actions The MCPWM operator is able to sense external signals with information about failure of the motor, the power driver or any other device connected. These failure signals are encapsulated into *MCPWM fault objects*.

You should determine possible failure modes of the motor and what action should be performed on detection of a particular fault, e.g., drive all outputs low for a brushed motor, lock current state for a stepper motor, etc. Because of this action, the motor should be put into a safe state to reduce the likelihood of damage caused by the fault.

Set Operator Brake Mode on Fault The way that MCPWM operator reacts to the fault is called **Brake**. The MCPWM operator can be configured to perform different brake modes for each fault object by calling *mcpwm_operator_set_brake_on_fault()*. Brake specific configuration is passed as a structure *mcpwm_brake_config_t*:

- *mcpwm_brake_config_t::fault* set which fault that the operator should react to.
- *mcpwm_brake_config_t::brake_mode* set the brake mode that should be used for the fault. The supported brake modes are listed in the *mcpwm_operator_brake_mode_t*. For *MCPWM_OPER_BRAKE_MODE_CBC* mode, the operator will recover itself automatically as long as the fault disappears. You can specify the recovery time in *mcpwm_brake_config_t::cbc_recover_on_tez* and *mcpwm_brake_config_t::cbc_recover_on_tep*. For *MCPWM_OPER_BRAKE_MODE_OST* mode, the operator can't recover even though the fault disappears. User has to call *mcpwm_operator_recover_from_fault()* to manually recover it.

Set Generator Action on Brake Event One generator can set multiple actions on different brake events, by calling *mcpwm_generator_set_actions_on_brake_event()* with variable number of action configurations. The action configuration is defined in *mcpwm_gen_brake_event_action_t*:

- *mcpwm_gen_brake_event_action_t::direction* specific the timer direction. The supported directions are listed in *mcpwm_timer_direction_t*.
- *mcpwm_gen_brake_event_action_t::brake_mode* specifies the brake mode. The supported brake modes are listed in the *mcpwm_operator_brake_mode_t*.
- *mcpwm_gen_brake_event_action_t::action* specifies the generator action to be taken. The supported actions are listed in *mcpwm_generator_action_t*.

There's a helper macro *MCPWM_GEN_BRAKE_EVENT_ACTION* to simplify the construction of a brake event action entry.

Please note, the argument list of *mcpwm_generator_set_actions_on_brake_event()* **must** be terminated by *MCPWM_GEN_BRAKE_EVENT_ACTION_END*.

You can also set the brake action one by one by calling *mcpwm_generator_set_action_on_brake_event()* without varargs.

Register Fault Event Callbacks The MCPWM fault detector can inform the user when it detects a valid fault or a fault signal disappears. If you have some function that should be called when such event happens, you should hook your function to the interrupt service routine by calling *mcpwm_fault_register_event_callbacks()*. The callback function prototype is declared in *mcpwm_fault_event_cb_t*. All supported event callbacks are listed in the *mcpwm_fault_event_callbacks_t*:

- *mcpwm_fault_event_callbacks_t::on_fault_enter* sets callback function that will be called when a fault is detected.
- *mcpwm_fault_event_callbacks_t::on_fault_exit* sets callback function that will be called when a fault is cleared.

The callback function is called within the ISR context, so is should **not** attempt to block (e.g., make sure that only FreeRTOS APIs with `ISR` suffix is called within the function).

The parameter `user_data` of *mcpwm_fault_register_event_callbacks()* function is used to save user's own context, it will be passed to the callback function directly.

This function will lazy install interrupt service for the MCPWM fault, whereas the service can only be removed in *mcpwm_del_fault*.

Register Brake Event Callbacks The MCPWM operator can inform the user when it going to take a brake action. If you have some function that should be called when this event happens, you should hook your function to the interrupt service routine by calling *mcpwm_operator_register_event_callbacks()*. The callback function prototype is declared in *mcpwm_brake_event_cb_t*. All supported event callbacks are listed in the *mcpwm_operator_event_callbacks_t*:

- *mcpwm_operator_event_callbacks_t::on_brake_cbc* sets callback function that will be called when the operator is going to take a *CBC* action.
- *mcpwm_operator_event_callbacks_t::on_brake_ost* sets callback function that will be called when the operator is going to take an *OST* action.

The callback function is called within the ISR context, so is should **not** attempt to block (e.g., make sure that only FreeRTOS APIs with `ISR` suffix is called within the function).

The parameter `user_data` of *mcpwm_operator_register_event_callbacks()* function is used to save user's own context, it will be passed to the callback function directly.

This function will lazy install interrupt service for the MCPWM operator, whereas the service can only be removed in *mcpwm_del_operator*.

Generator Force Actions Software can override generator output level at runtime, by calling *mcpwm_generator_set_force_level()*. The software force level always has a higher priority than other event actions set in e.g. *mcpwm_generator_set_actions_on_timer_event()*.

- Set the `level` to -1 means to disable the force action, and the generator's output level will be controlled by the event actions again.
- Set the `hold_on` to true, the force output level will keep alive, until it's removed by assigning `level` to -1.
- Set the `hole_on` to false, the force output level will only be active for a short time, any upcoming event can override it.

Synchronization When a sync signal is taken by the MCPWM timer, the timer will be forced into a pre-defined **phase**, where the phase is determined by count value and count direction. You can set the sync phase by calling *mcpwm_timer_set_phase_on_sync()*. The sync phase configuration is defined in *mcpwm_timer_sync_phase_config_t* structure:

- *mcpwm_timer_sync_phase_config_t::sync_src* sets the sync signal source. See *MCPWM Sync Sources* for how to create a sync source object. Specifically, if this is set to `NULL`, the driver will disable the sync feature for the MCPWM timer.
- *mcpwm_timer_sync_phase_config_t::count_value* sets the count value to load when the sync signal is taken.
- *mcpwm_timer_sync_phase_config_t::direction* sets the count direction when the sync signal is taken.

Likewise, the MCPWM capture timer *MCPWM Capture Timer* can be synced as well. You can set the sync phase for the capture timer by calling *mcpwm_capture_timer_set_phase_on_sync()*. The sync phase configuration is defined in *mcpwm_capture_timer_sync_phase_config_t* structure:

- *mcpwm_capture_timer_sync_phase_config_t::sync_src* sets the sync signal source. See *MCPWM Sync Sources* for how to create a sync source object. Specifically, if this is set to `NULL`, the driver will disable the sync feature for the MCPWM capture timer.
- *mcpwm_capture_timer_sync_phase_config_t::count_value* sets the count value to load when the sync signal is taken.
- *mcpwm_capture_timer_sync_phase_config_t::direction* sets the count direction when the sync signal is taken. Note that, different from MCPWM Timer, the capture timer can only support one count direction: *MCPWM_TIMER_DIRECTION_UP*.

Sync Timers by GPIO

```
static void example_setup_sync_strategy(mcpwm_timer_handle_t timers[])
{
    mcpwm_sync_handle_t gpio_sync_source = NULL;
    mcpwm_gpio_sync_src_config_t gpio_sync_config = {
        .group_id = 0,              // GPIO fault should be in the same group of␣
↪the above timers
        .gpio_num = EXAMPLE_SYNC_GPIO,
        .flags.pull_down = true,
```

(continues on next page)

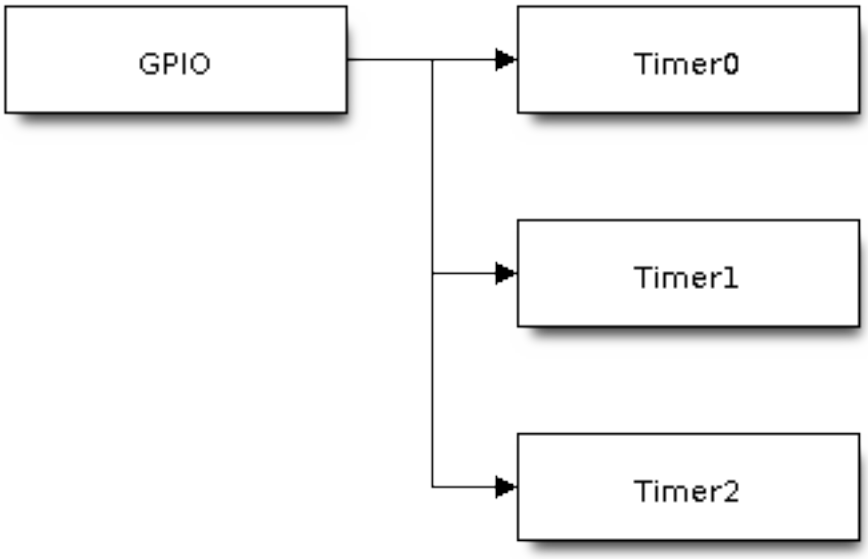


Fig. 13: GPIO Sync All MCPWM Timers

(continued from previous page)

```
        .flags.active_neg = false,  // by default, a posedge pulse can trigger a␣
↪sync event
    };
    ESP_ERROR_CHECK(mcpwm_new_gpio_sync_src(&gpio_sync_config, &gpio_sync_source));

    mcpwm_timer_sync_phase_config_t sync_phase_config = {
        .count_value = 0,                       // sync phase: target count value
        .direction = MCPWM_TIMER_DIRECTION_UP, // sync phase: count direction
        .sync_src = gpio_sync_source,           // sync source
    };
    for (int i = 0; i < 3; i++) {
        ESP_ERROR_CHECK(mcpwm_timer_set_phase_on_sync(timers[i], &sync_phase_
↪config));
    }
}
```

Capture The basic functionality of MCPWM capture is to record the time when any pulse edge of the capture signal turns active. Then you can get the pulse width and convert it into other physical quantities like distance or speed in the capture callback function. For example, in the BLDC (Brushless DC, see figure below) scenario, you can use the capture submodule to sense the rotor position from the Hall sensor.

The capture timer is usually connected with several capture channels, please refer to *MCPWM Capture Timer and Channels* for resource allocation.

Register Event Callbacks The MCPWM capture channel can inform the user when there's a valid edge detected on the signal. You have to register a callback function to get the timer count value of the capture moment, by calling `mcpwm_capture_channel_register_event_callbacks()`. The callback function prototype is declared in `mcpwm_capture_event_cb_t`. All supported capture callbacks are listed in the `mcpwm_capture_event_callbacks_t`:

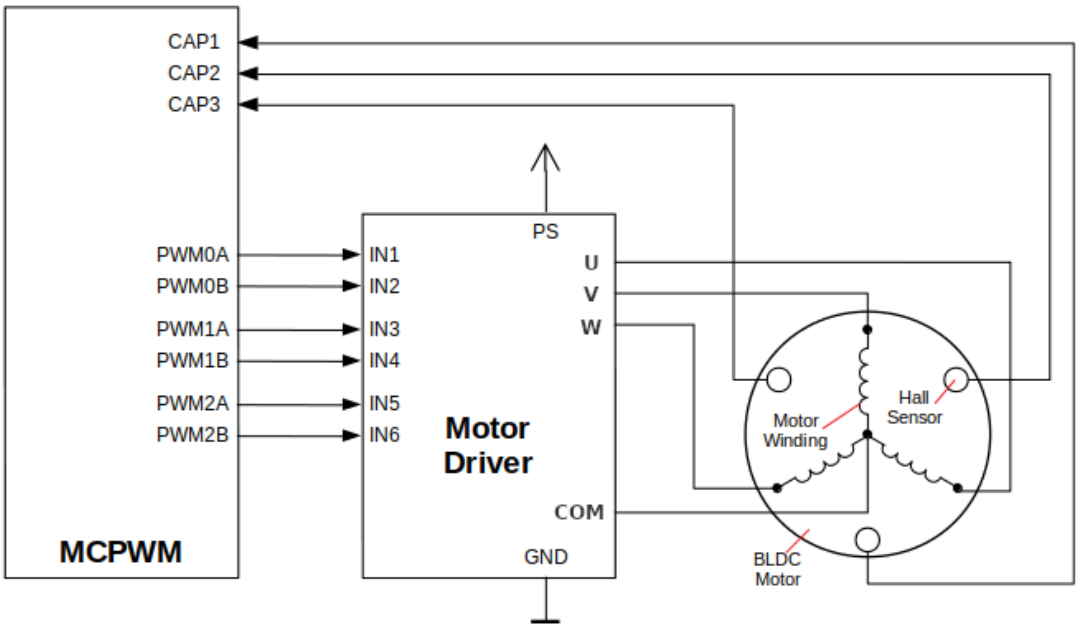


Fig. 14: MCPWM BLDC with Hall Sensor

- *mcpwm_capture_event_callbacks_t::on_cap* sets callback function for the capture channel when a valid edge is detected.

The callback function will provide event specific data of type *mcpwm_capture_event_data_t*, so that you can get the edge of the capture signal in *mcpwm_capture_event_data_t::cap_edge* and the count value of that moment in *mcpwm_capture_event_data_t::cap_value*. To convert the capture count into timestamp, you need to know the resolution of the capture timer by calling *mcpwm_capture_timer_get_resolution()*.

The callback function is called within the ISR context, so is should **not** attempt to block (e.g., make sure that only FreeRTOS APIs with `ISR` suffix is called within the function).

The parameter `user_data` of *mcpwm_capture_channel_register_event_callbacks()* function is used to save user's own context, it will be passed to the callback function directly.

This function will lazy install interrupt service for the MCPWM capture channel, whereas the service can only be removed in *mcpwm_del_capture_channel*.

Enable and Disable Capture Channel The capture channel is not enabled after allocation by *mcpwm_new_capture_channel()*. You should call *mcpwm_capture_channel_enable()* and *mcpwm_capture_channel_disable()* accordingly to enable or disable the channel. If the interrupt service is lazy installed during registering event callbacks for the channel in *mcpwm_capture_channel_register_event_callbacks()*, *mcpwm_capture_channel_enable()* will enable the interrupt service as well.

Enable and Disable Capture Timer Before doing IO control to the capture timer, user needs to enable the timer first, by calling *mcpwm_capture_timer_enable()*. Internally, this function will:

- switch the capture timer state from **init** to **enable**.
- acquire a proper power management lock if a specific clock source (e.g. APB clock) is selected. See also *Power management* for more information.

On the contrary, calling *mcpwm_capture_timer_disable()* will put the timer driver back to **init** state, and release the power management lock.

Start and Stop Capture Timer The basic IO operation of a capture timer is to start and stop. Calling *mcpwm_capture_timer_start()* can start the timer and calling *mcpwm_capture_timer_stop()* can stop the timer immediately.

Trigger a Software Capture Event Sometime, the software also wants to trigger a "fake" capture event. The *mcpwm_capture_channel_trigger_soft_catch()* is provided for that purpose. Please note that, even though it's a "fake" capture event, it can still cause an interrupt, thus your capture event callback function will get invoked as well.

Power Management When power management is enabled (i.e. *CONFIG_PM_ENABLE* is on), the system will adjust the PLL, APB frequency before going into light sleep, thus potentially changing the period of a MCPWM timers' counting step and leading to inaccurate time keeping.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type *ESP_PM_APB_FREQ_MAX*. Whenever the driver creates a MCPWM timer instance that has selected *MCPWM_TIMER_CLK_SRC_PLL160M* as its clock source, the driver will guarantee that the power management lock is acquired when enable the timer by *mcpwm_timer_enable()*. Likewise, the driver releases the lock when *mcpwm_timer_disable()* is called for that timer.

Likewise, Whenever the driver creates a MCPWM capture timer instance that has selected *MCPWM_CAPTURE_CLK_SRC_APB* as its clock source, the driver will guarantee that the power management lock is acquired when enable the timer by *mcpwm_capture_timer_enable()*. And will release the lock in *mcpwm_capture_timer_disable()*.

IRAM Safe By default, the MCPWM interrupt will be deferred when the Cache is disabled for reasons like writing/erasing Flash. Thus the event callback functions will not get executed in time, which is not expected in a real-time application.

There's a Kconfig option *CONFIG_MCPWM_ISR_IRAM_SAFE* that will:

1. Enable the interrupt being serviced even when cache is disabled
2. Place all functions that used by the ISR into IRAM[2]
3. Place driver object into DRAM (in case it's mapped to PSRAM by accident)

This will allow the interrupt to run while the cache is disabled but will come at the cost of increased IRAM consumption.

There is another Kconfig option *CONFIG_MCPWM_CTRL_FUNC_IN_IRAM* that can put commonly used IO control functions into IRAM as well. So, these functions can also be executable when the cache is disabled. These IO control functions are as follows:

- *mcpwm_comparator_set_compare_value()*
- *mcpwm_timer_set_period()*

Thread Safety The factory functions like *mcpwm_new_timer()* are guaranteed to be thread safe by the driver, which means, you can call it from different RTOS tasks without protection by extra locks.

The following functions are allowed to run under ISR context, as the driver uses a critical section to prevent them being called concurrently in the task and ISR.

- *mcpwm_comparator_set_compare_value()*
- *mcpwm_timer_set_period()*

Other functions that are not related to *Resource Allocation*, are not thread safe. Thus, you should avoid calling them in different tasks without mutex protection.

[2] Callback function and the sub-functions invoked by itself should also be placed in IRAM, users need to take care of this by themselves.

Kconfig Options

- *CONFIG_MCPWM_ISR_IRAM_SAFE* controls whether the default ISR handler can work when cache is disabled, see *IRAM Safe* for more information.
- *CONFIG_MCPWM_CTRL_FUNC_IN_IRAM* controls where to place the MCPWM control functions (IRAM or flash), see *IRAM Safe* for more information.
- *CONFIG_MCPWM_ENABLE_DEBUG_LOG* is used to enabled the debug log output. Enable this option will increase the firmware binary size.

Application Examples

- Brushed DC motor speed control by PID algorithm: peripherals/mcpwm/mcpwm_bdc_speed_control
- BLDC motor control with hall sensor feedback: peripherals/mcpwm/mcpwm_bldc_hall_control
- Ultrasonic sensor (HC-SR04) distance measurement: peripherals/mcpwm/mcpwm_capture_hc_sr04
- Servo motor angle control: peripherals/mcpwm/mcpwm_servo_control
- MCPWM synchronization between timers: peripherals/mcpwm/mcpwm_sync

API Reference

Header File

- components/driver/include/driver/mcpwm_timer.h

Functions

esp_err_t **mcpwm_new_timer**(const *mcpwm_timer_config_t* *config, *mcpwm_timer_handle_t* *ret_timer)

Create MCPWM timer.

Parameters
- **config** –**[in]** MCPWM timer configuration
- **ret_timer** –**[out]** Returned MCPWM timer handle

Returns
- ESP_OK: Create MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Create MCPWM timer failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM timer failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM timer failed because all hardware timers are used up and no more free one
- ESP_FAIL: Create MCPWM timer failed because of other error

esp_err_t **mcpwm_del_timer**(*mcpwm_timer_handle_t* timer)

Delete MCPWM timer.

Parameters **timer** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`

Returns
- ESP_OK: Delete MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete MCPWM timer failed because timer is not in init state
- ESP_FAIL: Delete MCPWM timer failed because of other error

esp_err_t **mcpwm_timer_set_period**(*mcpwm_timer_handle_t* timer, uint32_t period_ticks)

Set a new period for MCPWM timer.

Note: If *mcpwm_timer_config_t::update_period_on_empty* and *mcpwm_timer_config_t::update_period_on_sync* are not set, the new period will take effect immediately. Otherwise, the new period will take effect when timer counts to zero or on sync event.

Note: You may need to use `mcpwm_comparator_set_compare_value` to set a new compare value for MCPWM comparator in order to keep the same PWM duty cycle.

Parameters
- **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer`
- **`period_ticks`** –**[in]** New period in count ticks

Returns
- ESP_OK: Set new period for MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Set new period for MCPWM timer failed because of invalid argument
- ESP_FAIL: Set new period for MCPWM timer failed because of other error

esp_err_t **`mcpwm_timer_enable`**(*mcpwm_timer_handle_t* timer)

Enable MCPWM timer.

Parameters **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`

Returns
- ESP_OK: Enable MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Enable MCPWM timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable MCPWM timer failed because timer is enabled already
- ESP_FAIL: Enable MCPWM timer failed because of other error

esp_err_t **`mcpwm_timer_disable`**(*mcpwm_timer_handle_t* timer)

Disable MCPWM timer.

Parameters **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`

Returns
- ESP_OK: Disable MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Disable MCPWM timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable MCPWM timer failed because timer is disabled already
- ESP_FAIL: Disable MCPWM timer failed because of other error

esp_err_t **`mcpwm_timer_start_stop`**(*mcpwm_timer_handle_t* timer, *mcpwm_timer_start_stop_cmd_t* command)

Send specific start/stop commands to MCPWM timer.

Parameters
- **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`
- **`command`** –**[in]** Supported command list for MCPWM timer

Returns
- ESP_OK: Start or stop MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Start or stop MCPWM timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Start or stop MCPWM timer failed because timer is not enabled
- ESP_FAIL: Start or stop MCPWM timer failed because of other error

esp_err_t **`mcpwm_timer_register_event_callbacks`**(*mcpwm_timer_handle_t* timer, const *mcpwm_timer_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for MCPWM timer.

Note: The first call to this function needs to be before the call to `mcpwm_timer_enable`

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`
- **`cbs`** –**[in]** Group of callback functions
- **`user_data`** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because timer is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **`mcpwm_timer_set_phase_on_sync`**(*mcpwm_timer_handle_t* timer, const *mcpwm_timer_sync_phase_config_t* *config)

Set sync phase for MCPWM timer.

Parameters

- **`timer`** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`
- **`config`** –**[in]** MCPWM timer sync phase configuration

Returns

- ESP_OK: Set sync phase for MCPWM timer successfully
- ESP_ERR_INVALID_ARG: Set sync phase for MCPWM timer failed because of invalid argument
- ESP_FAIL: Set sync phase for MCPWM timer failed because of other error

Structures

struct **`mcpwm_timer_event_callbacks_t`**

Group of supported MCPWM timer event callbacks.

Note: The callbacks are all running under ISR environment

Public Members

mcpwm_timer_event_cb_t **`on_full`**

callback function when MCPWM timer counts to peak value

mcpwm_timer_event_cb_t **`on_empty`**

callback function when MCPWM timer counts to zero

mcpwm_timer_event_cb_t **`on_stop`**

callback function when MCPWM timer stops

struct **`mcpwm_timer_config_t`**

MCPWM timer configuration.

Public Members

int **group_id**

Specify from which group to allocate the MCPWM timer

mcpwm_timer_clock_source_t **clk_src**

MCPWM timer clock source

uint32_t **resolution_hz**

Counter resolution in Hz, ranges from around 300KHz to 80MHz. The step size of each count tick equals to (1 / resolution_hz) seconds

mcpwm_timer_count_mode_t **count_mode**

Count mode

uint32_t **period_ticks**

Number of count ticks within a period

int **intr_priority**

MCPWM timer interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **update_period_on_empty**

Whether to update period when timer counts to zero

uint32_t **update_period_on_sync**

Whether to update period on sync event

struct *mcpwm_timer_config_t*::[anonymous] **flags**

Extra configuration flags for timer

struct **mcpwm_timer_sync_phase_config_t**

MCPWM Timer sync phase configuration.

Public Members

mcpwm_sync_handle_t **sync_src**

The sync event source. Set to NULL will disable the timer being synced by others

uint32_t **count_value**

The count value that should lock to upon sync event

mcpwm_timer_direction_t **direction**

The count direction that should lock to upon sync event

Header File

- components/driver/include/driver/mcpwm_oper.h

Functions

esp_err_t **mcpwm_new_operator**(const *mcpwm_operator_config_t* *config, *mcpwm_oper_handle_t* *ret_oper)

Create MCPWM operator.

Parameters

- **config** –**[in]** MCPWM operator configuration
- **ret_oper** –**[out]** Returned MCPWM operator handle

Returns

- ESP_OK: Create MCPWM operator successfully
- ESP_ERR_INVALID_ARG: Create MCPWM operator failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM operator failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM operator failed because can't find free resource
- ESP_FAIL: Create MCPWM operator failed because of other error

esp_err_t **mcpwm_del_operator**(*mcpwm_oper_handle_t* oper)

Delete MCPWM operator.

Parameters **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`

Returns

- ESP_OK: Delete MCPWM operator successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM operator failed because of invalid argument
- ESP_FAIL: Delete MCPWM operator failed because of other error

esp_err_t **mcpwm_operator_connect_timer**(*mcpwm_oper_handle_t* oper, *mcpwm_timer_handle_t* timer)

Connect MCPWM operator and timer, so that the operator can be driven by the timer.

Parameters

- **oper** –**[in]** MCPWM operator handle, allocated by `mcpwm_new_operator()`
- **timer** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`

Returns

- ESP_OK: Connect MCPWM operator and timer successfully
- ESP_ERR_INVALID_ARG: Connect MCPWM operator and timer failed because of invalid argument
- ESP_FAIL: Connect MCPWM operator and timer failed because of other error

esp_err_t **mcpwm_operator_set_brake_on_fault**(*mcpwm_oper_handle_t* oper, const *mcpwm_brake_config_t* *config)

Set brake method for MCPWM operator.

Parameters

- **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`
- **config** –**[in]** MCPWM brake configuration

Returns

- ESP_OK: Set trip for operator successfully
- ESP_ERR_INVALID_ARG: Set trip for operator failed because of invalid argument
- ESP_FAIL: Set trip for operator failed because of other error

esp_err_t **mcpwm_operator_recover_from_fault**(*mcpwm_oper_handle_t* oper, *mcpwm_fault_handle_t* fault)

Try to make the operator recover from fault.

Note: To recover from fault or escape from trip, you make sure the fault signal has dissappeared already. Otherwise the recovery can't succeed.

Parameters

- **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`
- **fault** –**[in]** MCPWM fault handle

Returns

- ESP_OK: Recover from fault successfully
- ESP_ERR_INVALID_ARG: Recover from fault failed because of invalid argument
- ESP_ERR_INVALID_STATE: Recover from fault failed because the fault source is still active
- ESP_FAIL: Recover from fault failed because of other error

esp_err_t **mcpwm_operator_register_event_callbacks**(*mcpwm_oper_handle_t* oper, const *mcpwm_operator_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for MCPWM operator.

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **oper** –**[in]** MCPWM operator handle, allocated by `mcpwm_new_operator()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **mcpwm_operator_apply_carrier**(*mcpwm_oper_handle_t* oper, const *mcpwm_carrier_config_t* *config)

Apply carrier feature for MCPWM operator.

Parameters

- **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`
- **config** –**[in]** MCPWM carrier specific configuration

Returns

- ESP_OK: Set carrier for operator successfully
- ESP_ERR_INVALID_ARG: Set carrier for operator failed because of invalid argument
- ESP_FAIL: Set carrier for operator failed because of other error

Structures

struct **mcpwm_operator_config_t**

MCPWM operator configuration.

Public Members

int **group_id**

Specify from which group to allocate the MCPWM operator

int **intr_priority**

MCPWM operator interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **update_gen_action_on_tez**

Whether to update generator action when timer counts to zero

uint32_t **update_gen_action_on_tep**

Whether to update generator action when timer counts to peak

uint32_t **update_gen_action_on_sync**

Whether to update generator action on sync event

uint32_t **update_dead_time_on_tez**

Whether to update dead time when timer counts to zero

uint32_t **update_dead_time_on_tep**

Whether to update dead time when timer counts to peak

uint32_t **update_dead_time_on_sync**

Whether to update dead time on sync event

struct *mcpwm_operator_config_t*::[anonymous] **flags**

Extra configuration flags for operator

struct **mcpwm_brake_config_t**

MCPWM brake configuration structure.

Public Members

mcpwm_fault_handle_t **fault**

Which fault causes the operator to brake

mcpwm_operator_brake_mode_t **brake_mode**

Brake mode

uint32_t **cbc_recover_on_tez**

Recovery CBC brake state on tez event

uint32_t **cbc_recover_on_tep**

Recovery CBC brake state on tep event

struct *mcpwm_brake_config_t*::[anonymous] **flags**

Extra flags for brake configuration

struct **mcpwm_operator_event_callbacks_t**

Group of supported MCPWM operator event callbacks.

Note: The callbacks are all running under ISR environment

Public Members

mcpwm_brake_event_cb_t **on_brake_cbc**

callback function when mcpwm operator brakes in CBC

mcpwm_brake_event_cb_t **on_brake_ost**

callback function when mcpwm operator brakes in OST

struct **mcpwm_carrier_config_t**

MCPWM carrier configuration structure.

Public Members

mcpwm_carrier_clock_source_t **clk_src**

MCPWM carrier clock source

uint32_t **frequency_hz**

Carrier frequency in Hz

uint32_t **first_pulse_duration_us**

The duration of the first PWM pulse, in us

float **duty_cycle**

Carrier duty cycle

uint32_t **invert_before_modulate**

Invert the raw signal

uint32_t **invert_after_modulate**

Invert the modulated signal

struct *mcpwm_carrier_config_t*::[anonymous] **flags**

Extra flags for carrier configuration

Header File

- components/driver/include/driver/mcpwm_cmpr.h

Functions

esp_err_t **mcpwm_new_comparator**(*mcpwm_oper_handle_t* oper, const *mcpwm_comparator_config_t* *config, *mcpwm_cmpr_handle_t* *ret_cmpr)

Create MCPWM comparator.

Parameters

- **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`, the new comparator will be allocated from this operator
- **config** –**[in]** MCPWM comparator configuration
- **ret_cmpr** –**[out]** Returned MCPWM comparator

Returns

- ESP_OK: Create MCPWM comparator successfully
- ESP_ERR_INVALID_ARG: Create MCPWM comparator failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM comparator failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM comparator failed because can't find free resource
- ESP_FAIL: Create MCPWM comparator failed because of other error

esp_err_t **mcpwm_del_comparator**(*mcpwm_cmpr_handle_t* cmpr)

Delete MCPWM comparator.

Parameters **cmpr** –**[in]** MCPWM comparator handle, allocated by `mcpwm_new_comparator()`

Returns

- ESP_OK: Delete MCPWM comparator successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM comparator failed because of invalid argument
- ESP_FAIL: Delete MCPWM comparator failed because of other error

esp_err_t **mcpwm_comparator_register_event_callbacks**(*mcpwm_cmpr_handle_t* cmpr, const *mcpwm_comparator_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for MCPWM comparator.

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **cmpr** –**[in]** MCPWM comparator handle, allocated by `mcpwm_new_comparator()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **mcpwm_comparator_set_compare_value**(*mcpwm_cmpr_handle_t* cmpr, uint32_t cmp_ticks)

Set MCPWM comparator's compare value.

Parameters

- **cmpr** –**[in]** MCPWM comparator handle, allocated by `mcpwm_new_comparator()`
- **cmp_ticks** –**[in]** The new compare value

Returns

- ESP_OK: Set MCPWM compare value successfully
- ESP_ERR_INVALID_ARG: Set MCPWM compare value failed because of invalid argument (e.g. the cmp_ticks is out of range)
- ESP_ERR_INVALID_STATE: Set MCPWM compare value failed because the operator doesn't have a timer connected
- ESP_FAIL: Set MCPWM compare value failed because of other error

Structures

struct **mcpwm_comparator_config_t**

MCPWM comparator configuration.

Public Members

int **intr_priority**

MCPWM comparator interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **update_cmp_on_tez**

Whether to update compare value when timer count equals to zero (tez)

uint32_t **update_cmp_on_tep**

Whether to update compare value when timer count equals to peak (tep)

uint32_t **update_cmp_on_sync**

Whether to update compare value on sync event

struct *mcpwm_comparator_config_t*::[anonymous] **flags**

Extra configuration flags for comparator

struct **mcpwm_comparator_event_callbacks_t**

Group of supported MCPWM compare event callbacks.

Note: The callbacks are all running under ISR environment

Public Members

mcpwm_compare_event_cb_t **on_reach**

ISR callback function which would be invoked when counter reaches compare value

Header File

- components/driver/include/driver/mcpwm_gen.h

Functions

esp_err_t **mcpwm_new_generator**(*mcpwm_oper_handle_t* oper, const *mcpwm_generator_config_t* *config, *mcpwm_gen_handle_t* *ret_gen)

Allocate MCPWM generator from given operator.

Parameters

- **oper** –**[in]** MCPWM operator, allocated by `mcpwm_new_operator()`
- **config** –**[in]** MCPWM generator configuration
- **ret_gen** –**[out]** Returned MCPWM generator

Returns

- ESP_OK: Create MCPWM generator successfully
- ESP_ERR_INVALID_ARG: Create MCPWM generator failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM generator failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM generator failed because can't find free resource
- ESP_FAIL: Create MCPWM generator failed because of other error

esp_err_t **mcpwm_del_generator**(*mcpwm_gen_handle_t* gen)

Delete MCPWM generator.

Parameters **gen** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`

Returns

- ESP_OK: Delete MCPWM generator successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM generator failed because of invalid argument
- ESP_FAIL: Delete MCPWM generator failed because of other error

esp_err_t **mcpwm_generator_set_force_level** (*mcpwm_gen_handle_t* gen, int level, bool hold_on)

Set force level for MCPWM generator.

Note: The force level will be applied to the generator immediately, regardless any other events that would change the generator's behaviour.

Note: If the `hold_on` is true, the force level will retain forever, until user removes the force level by setting the force level to `-1`.

Note: If the `hold_on` is false, the force level can be overridden by the next event action.

Note: The force level set by this function can be inverted by GPIO matrix or dead-time module. So the level set here doesn't equal to the final output level.

Parameters

- **gen** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **level** –**[in]** GPIO level to be applied to MCPWM generator, specially, -1 means to remove the force level
- **hold_on** –**[in]** Whether the forced PWM level should retain (i.e. will remain unchanged until manually remove the force level)

Returns

- ESP_OK: Set force level for MCPWM generator successfully
- ESP_ERR_INVALID_ARG: Set force level for MCPWM generator failed because of invalid argument
- ESP_FAIL: Set force level for MCPWM generator failed because of other error

esp_err_t **mcpwm_generator_set_action_on_timer_event** (*mcpwm_gen_handle_t* gen, *mcpwm_gen_timer_event_action_t* ev_act)

Set generator action on MCPWM timer event.

Parameters

- **gen** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –**[in]** MCPWM timer event action, can be constructed by `MCPWM_GEN_TIMER_EVENT_ACTION` helper macro

Returns

- ESP_OK: Set generator action successfully
- ESP_ERR_INVALID_ARG: Set generator action failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set generator action failed because of timer is not connected to operator
- ESP_FAIL: Set generator action failed because of other error

esp_err_t **mcpwm_generator_set_actions_on_timer_event** (*mcpwm_gen_handle_t* gen, *mcpwm_gen_timer_event_action_t* ev_act, ...)

Set generator actions on multiple MCPWM timer events.

Note: This is an aggregation version of `mcpwm_generator_set_action_on_timer_event`, which allows user to set multiple actions in one call.

Parameters

- **gen** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –**[in]** MCPWM timer event action list, must be terminated by `MCPWM_GEN_TIMER_EVENT_ACTION_END()`

Returns

- ESP_OK: Set generator actions successfully
- ESP_ERR_INVALID_ARG: Set generator actions failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set generator actions failed because of timer is not connected to operator
- ESP_FAIL: Set generator actions failed because of other error

esp_err_t **mcpwm_generator_set_action_on_compare_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_compare_event_action_t* ev_act)

Set generator action on MCPWM compare event.

Parameters

- **generator** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –**[in]** MCPWM compare event action, can be constructed by `MCPWM_GEN_COMPARE_EVENT_ACTION` helper macro

Returns

- ESP_OK: Set generator action successfully
- ESP_ERR_INVALID_ARG: Set generator action failed because of invalid argument
- ESP_FAIL: Set generator action failed because of other error

esp_err_t **mcpwm_generator_set_actions_on_compare_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_compare_event_action_t* ev_act, ...)

Set generator actions on multiple MCPWM compare events.

Note: This is an aggregation version of `mcpwm_generator_set_action_on_compare_event`, which allows user to set multiple actions in one call.

Parameters

- **generator** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –**[in]** MCPWM compare event action list, must be terminated by `MCPWM_GEN_COMPARE_EVENT_ACTION_END()`

Returns

- ESP_OK: Set generator actions successfully
- ESP_ERR_INVALID_ARG: Set generator actions failed because of invalid argument
- ESP_FAIL: Set generator actions failed because of other error

esp_err_t **mcpwm_generator_set_action_on_brake_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_brake_event_action_t* ev_act)

Set generator action on MCPWM brake event.

Parameters

- **generator** –**[in]** MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –**[in]** MCPWM brake event action, can be constructed by `MCPWM_GEN_BRAKE_EVENT_ACTION` helper macro

Returns

- ESP_OK: Set generator action successfully
- ESP_ERR_INVALID_ARG: Set generator action failed because of invalid argument
- ESP_FAIL: Set generator action failed because of other error

esp_err_t **mcpwm_generator_set_actions_on_brake_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_brake_event_action_t* ev_act, ...)

Set generator actions on multiple MCPWM brake events.

Note: This is an aggregation version of `mcpwm_generator_set_action_on_brake_event`, which allows user to set multiple actions in one call.

Parameters

- **generator** –[in] MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –[in] MCPWM brake event action list, must be terminated by `MCPWM_GEN_BRAKE_EVENT_ACTION_END()`

Returns

- ESP_OK: Set generator actions successfully
- ESP_ERR_INVALID_ARG: Set generator actions failed because of invalid argument
- ESP_FAIL: Set generator actions failed because of other error

esp_err_t **mcpwm_generator_set_action_on_fault_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_fault_event_action_t* ev_act)

Set generator action on MCPWM Fault event.

Parameters

- **generator** –[in] MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –[in] MCPWM trigger event action, can be constructed by `MCPWM_GEN_FAULT_EVENT_ACTION` helper macro

Returns

- ESP_OK: Set generator action successfully
- ESP_ERR_INVALID_ARG: Set generator action failed because of invalid argument
- ESP_FAIL: Set generator action failed because of other error

esp_err_t **mcpwm_generator_set_action_on_sync_event**(*mcpwm_gen_handle_t* generator, *mcpwm_gen_sync_event_action_t* ev_act)

Set generator action on MCPWM Sync event.

Note: The trigger only support one sync action, regardless of the kinds. Should not call this function more than once.

Parameters

- **generator** –[in] MCPWM generator handle, allocated by `mcpwm_new_generator()`
- **ev_act** –[in] MCPWM trigger event action, can be constructed by `MCPWM_GEN_SYNC_EVENT_ACTION` helper macro

Returns

- ESP_OK: Set generator action successfully
- ESP_ERR_INVALID_ARG: Set generator action failed because of invalid argument
- ESP_FAIL: Set generator action failed because of other error

esp_err_t **mcpwm_generator_set_dead_time**(*mcpwm_gen_handle_t* in_generator, *mcpwm_gen_handle_t* out_generator, const *mcpwm_dead_time_config_t* *config)

Set dead time for MCPWM generator.

Note: Due to a hardware limitation, you can't set rising edge delay for both MCPWM generator 0 and 1 at the same time, otherwise, there will be a conflict inside the dead time module. The same goes for the falling edge setting. But you can set both the rising edge and falling edge delay for the same MCPWM generator.

Parameters

- **in_generator** –**[in]** MCPWM generator, before adding the dead time
- **out_generator** –**[in]** MCPWM generator, after adding the dead time
- **config** –**[in]** MCPWM dead time configuration

Returns

- ESP_OK: Set dead time for MCPWM generator successfully
- ESP_ERR_INVALID_ARG: Set dead time for MCPWM generator failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set dead time for MCPWM generator failed because of invalid state (e.g. delay module is already in use by other generator)
- ESP_FAIL: Set dead time for MCPWM generator failed because of other error

Structures

struct **mcpwm_generator_config_t**

MCPWM generator configuration.

Public Members

int **gen_gpio_num**

The GPIO number used to output the PWM signal

uint32_t **invert_pwm**

Whether to invert the PWM signal (done by GPIO matrix)

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

uint32_t **io_od_mode**

Configure the GPIO as open-drain mode

uint32_t **pull_up**

Whether to pull up internally

uint32_t **pull_down**

Whether to pull down internally

struct *mcpwm_generator_config_t*::[anonymous] **flags**

Extra configuration flags for generator

struct **mcpwm_gen_timer_event_action_t**

Generator action on specific timer event.

Public Members

mcpwm_timer_direction_t **direction**

Timer direction

mcpwm_timer_event_t **event**

Timer event

mcpwm_generator_action_t **action**

Generator action should perform

struct **mcpwm_gen_compare_event_action_t**

Generator action on specific comparator event.

Public Members

mcpwm_timer_direction_t **direction**

Timer direction

mcpwm_cmpr_handle_t **comparator**

Comparator handle

mcpwm_generator_action_t **action**

Generator action should perform

struct **mcpwm_gen_brake_event_action_t**

Generator action on specific brake event.

Public Members

mcpwm_timer_direction_t **direction**

Timer direction

mcpwm_operator_brake_mode_t **brake_mode**

Brake mode

mcpwm_generator_action_t **action**

Generator action should perform

struct **mcpwm_gen_fault_event_action_t**

Generator action on specific fault event.

Public Members

mcpwm_timer_direction_t **direction**

Timer direction

mcpwm_fault_handle_t **fault**

Which fault as the trigger. Only support GPIO fault

mcpwm_generator_action_t **action**

Generator action should perform

struct **mcpwm_gen_sync_event_action_t**

Generator action on specific sync event.

Public Members

mcpwm_timer_direction_t **direction**

Timer direction

mcpwm_sync_handle_t **sync**

Which sync as the trigger

mcpwm_generator_action_t **action**

Generator action should perform

struct **mcpwm_dead_time_config_t**

MCPWM dead time configuration structure.

Public Members

uint32_t **posedge_delay_ticks**

delay time applied to rising edge, 0 means no rising delay time

uint32_t **negedge_delay_ticks**

delay time applied to falling edge, 0 means no falling delay time

uint32_t **invert_output**

Invert the signal after applied the dead time

struct *mcpwm_dead_time_config_t*::[anonymous] **flags**

Extra flags for dead time configuration

Macros

MCPWM_GEN_TIMER_EVENT_ACTION(dir, ev, act)

Help macros to construct a *mcpwm_gen_timer_event_action_t* entry.

MCPWM_GEN_TIMER_EVENT_ACTION_END()

MCPWM_GEN_COMPARE_EVENT_ACTION(dir, cmp, act)

Help macros to construct a *mcpwm_gen_compare_event_action_t* entry.

MCPWM_GEN_COMPARE_EVENT_ACTION_END()

MCPWM_GEN_BRAKE_EVENT_ACTION(dir, mode, act)

Help macros to construct a *mcpwm_gen_brake_event_action_t* entry.

MCPWM_GEN_BRAKE_EVENT_ACTION_END()

MCPWM_GEN_FAULT_EVENT_ACTION(dir, flt, act)

Help macros to construct a *mcpwm_gen_fault_event_action_t* entry.

MCPWM_GEN_SYNC_EVENT_ACTION(dir, syn, act)

Help macros to construct a *mcpwm_gen_sync_event_action_t* entry.

Header File

- components/driver/include/driver/mcpwm_fault.h

Functions

esp_err_t **mcpwm_new_gpio_fault**(const *mcpwm_gpio_fault_config_t* *config, *mcpwm_fault_handle_t* *ret_fault)

Create MCPWM GPIO fault.

Parameters

- **config** –**[in]** MCPWM GPIO fault configuration
- **ret_fault** –**[out]** Returned GPIO fault handle

Returns

- ESP_OK: Create MCPWM GPIO fault successfully
- ESP_ERR_INVALID_ARG: Create MCPWM GPIO fault failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM GPIO fault failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM GPIO fault failed because can't find free resource
- ESP_FAIL: Create MCPWM GPIO fault failed because of other error

esp_err_t **mcpwm_new_soft_fault**(const *mcpwm_soft_fault_config_t* *config, *mcpwm_fault_handle_t* *ret_fault)

Create MCPWM software fault.

Parameters

- **config** –**[in]** MCPWM software fault configuration
- **ret_fault** –**[out]** Returned software fault handle

Returns

- ESP_OK: Create MCPWM software fault successfully
- ESP_ERR_INVALID_ARG: Create MCPWM software fault failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM software fault failed because out of memory
- ESP_FAIL: Create MCPWM software fault failed because of other error

esp_err_t **mcpwm_del_fault**(*mcpwm_fault_handle_t* fault)

Delete MCPWM fault.

Parameters **fault** –**[in]** MCPWM fault handle allocated by `mcpwm_new_gpio_fault()` or `mcpwm_new_soft_fault()`

Returns

- ESP_OK: Delete MCPWM fault successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM fault failed because of invalid argument
- ESP_FAIL: Delete MCPWM fault failed because of other error

esp_err_t **mcpwm_soft_fault_activate**(*mcpwm_fault_handle_t* fault)

Activate the software fault, trigger the fault event for once.

Parameters **fault** –**[in]** MCPWM soft fault, allocated by `mcpwm_new_soft_fault()`

Returns

- ESP_OK: Trigger MCPWM software fault event successfully
- ESP_ERR_INVALID_ARG: Trigger MCPWM software fault event failed because of invalid argument

- ESP_FAIL: Trigger MCPWM software fault event failed because of other error

esp_err_t **mcpwm_fault_register_event_callbacks**(*mcpwm_fault_handle_t* fault, const *mcpwm_fault_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for MCPWM fault.

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **fault** –**[in]** MCPWM GPIO fault handle, allocated by `mcpwm_new_gpio_fault()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

Structures

struct **mcpwm_gpio_fault_config_t**

MCPWM GPIO fault configuration structure.

Public Members

int **group_id**

In which MCPWM group that the GPIO fault belongs to

int **intr_priority**

MCPWM GPIO fault interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

int **gpio_num**

GPIO used by the fault signal

uint32_t **active_level**

On which level the fault signal is treated as active

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

uint32_t **pull_up**

Whether to pull up internally

uint32_t **pull_down**

Whether to pull down internally

struct *mcpwm_gpio_fault_config_t*::[anonymous] **flags**

Extra configuration flags for GPIO fault

struct **mcpwm_soft_fault_config_t**

MCPWM software fault configuration structure.

struct **mcpwm_fault_event_callbacks_t**

Group of supported MCPWM fault event callbacks.

Note: The callbacks are all running under ISR environment

Public Members

mcpwm_fault_event_cb_t **on_fault_enter**

ISR callback function that would be invoked when fault signal becomes active

mcpwm_fault_event_cb_t **on_fault_exit**

ISR callback function that would be invoked when fault signal becomes inactive

Header File

- components/driver/include/driver/mcpwm_sync.h

Functions

esp_err_t **mcpwm_new_timer_sync_src**(*mcpwm_timer_handle_t* timer, const *mcpwm_timer_sync_src_config_t* *config, *mcpwm_sync_handle_t* *ret_sync)

Create MCPWM timer sync source.

Parameters

- **timer** –**[in]** MCPWM timer handle, allocated by `mcpwm_new_timer()`
- **config** –**[in]** MCPWM timer sync source configuration
- **ret_sync** –**[out]** Returned MCPWM sync handle

Returns

- ESP_OK: Create MCPWM timer sync source successfully
- ESP_ERR_INVALID_ARG: Create MCPWM timer sync source failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM timer sync source failed because out of memory
- ESP_ERR_INVALID_STATE: Create MCPWM timer sync source failed because the timer has created a sync source before
- ESP_FAIL: Create MCPWM timer sync source failed because of other error

esp_err_t **mcpwm_new_gpio_sync_src**(const *mcpwm_gpio_sync_src_config_t* *config, *mcpwm_sync_handle_t* *ret_sync)

Create MCPWM GPIO sync source.

Parameters

- **config** –**[in]** MCPWM GPIO sync source configuration
- **ret_sync** –**[out]** Returned MCPWM GPIO sync handle

Returns

- ESP_OK: Create MCPWM GPIO sync source successfully
- ESP_ERR_INVALID_ARG: Create MCPWM GPIO sync source failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM GPIO sync source failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM GPIO sync source failed because can't find free resource

- ESP_FAIL: Create MCPWM GPIO sync source failed because of other error

esp_err_t **mcpwm_new_soft_sync_src**(const *mcpwm_soft_sync_config_t* *config, *mcpwm_sync_handle_t* *ret_sync)

Create MCPWM software sync source.

Parameters
- **config** –**[in]** MCPWM software sync source configuration
- **ret_sync** –**[out]** Returned software sync handle

Returns
- ESP_OK: Create MCPWM software sync successfully
- ESP_ERR_INVALID_ARG: Create MCPWM software sync failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM software sync failed because out of memory
- ESP_FAIL: Create MCPWM software sync failed because of other error

esp_err_t **mcpwm_del_sync_src**(*mcpwm_sync_handle_t* sync)

Delete MCPWM sync source.

Parameters **sync** –**[in]** MCPWM sync handle, allocated by `mcpwm_new_timer_sync_src()` or `mcpwm_new_gpio_sync_src()` or `mcpwm_new_soft_sync_src()`

Returns
- ESP_OK: Delete MCPWM sync source successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM sync source failed because of invalid argument
- ESP_FAIL: Delete MCPWM sync source failed because of other error

esp_err_t **mcpwm_soft_sync_activate**(*mcpwm_sync_handle_t* sync)

Activate the software sync, trigger the sync event for once.

Parameters **sync** –**[in]** MCPWM soft sync handle, allocated by `mcpwm_new_soft_sync_src()`

Returns
- ESP_OK: Trigger MCPWM software sync event successfully
- ESP_ERR_INVALID_ARG: Trigger MCPWM software sync event failed because of invalid argument
- ESP_FAIL: Trigger MCPWM software sync event failed because of other error

Structures

struct **mcpwm_timer_sync_src_config_t**

MCPWM timer sync source configuration.

Public Members

mcpwm_timer_event_t **timer_event**

Timer event, upon which MCPWM timer will generate the sync signal

uint32_t **propagate_input_sync**

The input sync signal would be routed to its sync output

struct *mcpwm_timer_sync_src_config_t*::[anonymous] **flags**

Extra configuration flags for timer sync source

struct **mcpwm_gpio_sync_src_config_t**

MCPWM GPIO sync source configuration.

Public Members

int **group_id**

MCPWM group ID

int **gpio_num**

GPIO used by sync source

uint32_t **active_neg**

Whether the sync signal is active on negedge, by default, the sync signal' s posedge is treated as active

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

uint32_t **pull_up**

Whether to pull up internally

uint32_t **pull_down**

Whether to pull down internally

struct *mcpwm_gpio_sync_src_config_t*::[anonymous] **flags**

Extra configuration flags for GPIO sync source

struct **mcpwm_soft_sync_config_t**

MCPWM software sync configuration structure.

Header File

- components/driver/include/driver/mcpwm_cap.h

Functions

esp_err_t **mcpwm_new_capture_timer**(const *mcpwm_capture_timer_config_t* *config, *mcpwm_cap_timer_handle_t* *ret_cap_timer)

Create MCPWM capture timer.

Parameters

- **config** –**[in]** MCPWM capture timer configuration
- **ret_cap_timer** –**[out]** Returned MCPWM capture timer handle

Returns

- ESP_OK: Create MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Create MCPWM capture timer failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM capture timer failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM capture timer failed because can't find free resource
- ESP_FAIL: Create MCPWM capture timer failed because of other error

esp_err_t **mcpwm_del_capture_timer**(*mcpwm_cap_timer_handle_t* cap_timer)

Delete MCPWM capture timer.

Parameters **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`

Returns

- ESP_OK: Delete MCPWM capture timer successfully

- ESP_ERR_INVALID_ARG: Delete MCPWM capture timer failed because of invalid argument
- ESP_FAIL: Delete MCPWM capture timer failed because of other error

esp_err_t **mcpwm_capture_timer_enable**(*mcpwm_cap_timer_handle_t* cap_timer)

Enable MCPWM capture timer.

Parameters **cap_timer** –**[in]** MCPWM capture timer handle, allocated by `mcpwm_new_capture_timer()`

Returns

- ESP_OK: Enable MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Enable MCPWM capture timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable MCPWM capture timer failed because timer is enabled already
- ESP_FAIL: Enable MCPWM capture timer failed because of other error

esp_err_t **mcpwm_capture_timer_disable**(*mcpwm_cap_timer_handle_t* cap_timer)

Disable MCPWM capture timer.

Parameters **cap_timer** –**[in]** MCPWM capture timer handle, allocated by `mcpwm_new_capture_timer()`

Returns

- ESP_OK: Disable MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Disable MCPWM capture timer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable MCPWM capture timer failed because timer is disabled already
- ESP_FAIL: Disable MCPWM capture timer failed because of other error

esp_err_t **mcpwm_capture_timer_start**(*mcpwm_cap_timer_handle_t* cap_timer)

Start MCPWM capture timer.

Parameters **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`

Returns

- ESP_OK: Start MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Start MCPWM capture timer failed because of invalid argument
- ESP_FAIL: Start MCPWM capture timer failed because of other error

esp_err_t **mcpwm_capture_timer_stop**(*mcpwm_cap_timer_handle_t* cap_timer)

Stop MCPWM capture timer.

Parameters **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`

Returns

- ESP_OK: Stop MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Stop MCPWM capture timer failed because of invalid argument
- ESP_FAIL: Stop MCPWM capture timer failed because of other error

esp_err_t **mcpwm_capture_timer_get_resolution**(*mcpwm_cap_timer_handle_t* cap_timer, uint32_t *out_resolution)

Get MCPWM capture timer resolution, in Hz.

Parameters

- **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`
- **out_resolution** –**[out]** Returned capture timer resolution, in Hz

Returns

- ESP_OK: Get capture timer resolution successfully

- ESP_ERR_INVALID_ARG: Get capture timer resolution failed because of invalid argument
- ESP_FAIL: Get capture timer resolution failed because of other error

esp_err_t **mcpwm_capture_timer_set_phase_on_sync**(*mcpwm_cap_timer_handle_t* cap_timer, const *mcpwm_capture_timer_sync_phase_config_t* *config)

Set sync phase for MCPWM capture timer.

Parameters

- **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`
- **config** –**[in]** MCPWM capture timer sync phase configuration

Returns

- ESP_OK: Set sync phase for MCPWM capture timer successfully
- ESP_ERR_INVALID_ARG: Set sync phase for MCPWM capture timer failed because of invalid argument
- ESP_FAIL: Set sync phase for MCPWM capture timer failed because of other error

esp_err_t **mcpwm_new_capture_channel**(*mcpwm_cap_timer_handle_t* cap_timer, const *mcpwm_capture_channel_config_t* *config, *mcpwm_cap_channel_handle_t* *ret_cap_channel)

Create MCPWM capture channel.

Note: The created capture channel won't be enabled until calling `mcpwm_capture_channel_enable`

Parameters

- **cap_timer** –**[in]** MCPWM capture timer, allocated by `mcpwm_new_capture_timer()`, will be connected to the new capture channel
- **config** –**[in]** MCPWM capture channel configuration
- **ret_cap_channel** –**[out]** Returned MCPWM capture channel

Returns

- ESP_OK: Create MCPWM capture channel successfully
- ESP_ERR_INVALID_ARG: Create MCPWM capture channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create MCPWM capture channel failed because out of memory
- ESP_ERR_NOT_FOUND: Create MCPWM capture channel failed because can't find free resource
- ESP_FAIL: Create MCPWM capture channel failed because of other error

esp_err_t **mcpwm_del_capture_channel**(*mcpwm_cap_channel_handle_t* cap_channel)

Delete MCPWM capture channel.

Parameters **cap_channel** –**[in]** MCPWM capture channel handle, allocated by `mcpwm_new_capture_channel()`

Returns

- ESP_OK: Delete MCPWM capture channel successfully
- ESP_ERR_INVALID_ARG: Delete MCPWM capture channel failed because of invalid argument
- ESP_FAIL: Delete MCPWM capture channel failed because of other error

esp_err_t **mcpwm_capture_channel_enable**(*mcpwm_cap_channel_handle_t* cap_channel)

Enable MCPWM capture channel.

Note: This function will transit the channel state from init to enable.

Note: This function will enable the interrupt service, if it's lazy installed in `mcpwm_capture_channel_register_event_callbacks()`.

Parameters **`cap_channel`** –**[in]** MCPWM capture channel handle, allocated by `mcpwm_new_capture_channel()`

Returns

- ESP_OK: Enable MCPWM capture channel successfully
- ESP_ERR_INVALID_ARG: Enable MCPWM capture channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable MCPWM capture channel failed because the channel is already enabled
- ESP_FAIL: Enable MCPWM capture channel failed because of other error

esp_err_t **`mcpwm_capture_channel_disable`**(*mcpwm_cap_channel_handle_t* cap_channel)

Disable MCPWM capture channel.

Parameters **`cap_channel`** –**[in]** MCPWM capture channel handle, allocated by `mcpwm_new_capture_channel()`

Returns

- ESP_OK: Disable MCPWM capture channel successfully
- ESP_ERR_INVALID_ARG: Disable MCPWM capture channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable MCPWM capture channel failed because the channel is not enabled yet
- ESP_FAIL: Disable MCPWM capture channel failed because of other error

esp_err_t **`mcpwm_capture_channel_register_event_callbacks`**(*mcpwm_cap_channel_handle_t* cap_channel, const *mcpwm_capture_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for MCPWM capture channel.

Note: The first call to this function needs to be before the call to `mcpwm_capture_channel_enable`

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **`cap_channel`** –**[in]** MCPWM capture channel handle, allocated by `mcpwm_new_capture_channel()`
- **`cbs`** –**[in]** Group of callback functions
- **`user_data`** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because the channel is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **`mcpwm_capture_channel_trigger_soft_catch`**(*mcpwm_cap_channel_handle_t* cap_channel)

Trigger a catch by software.

Parameters **cap_channel** –**[in]** MCPWM capture channel handle, allocated by `mcpwm_new_capture_channel()`

Returns

- ESP_OK: Trigger software catch successfully
- ESP_ERR_INVALID_ARG: Trigger software catch failed because of invalid argument
- ESP_ERR_INVALID_STATE: Trigger software catch failed because the channel is not enabled yet
- ESP_FAIL: Trigger software catch failed because of other error

Structures

struct **mcpwm_capture_timer_config_t**

MCPWM capture timer configuration structure.

Public Members

int **group_id**

Specify from which group to allocate the capture timer

mcpwm_capture_clock_source_t **clk_src**

MCPWM capture timer clock source

struct **mcpwm_capture_timer_sync_phase_config_t**

MCPWM Capture timer sync phase configuration.

Public Members

mcpwm_sync_handle_t **sync_src**

The sync event source

uint32_t **count_value**

The count value that should lock to upon sync event

mcpwm_timer_direction_t **direction**

The count direction that should lock to upon sync event

struct **mcpwm_capture_channel_config_t**

MCPWM capture channel configuration structure.

Public Members

int **gpio_num**

GPIO used capturing input signal

int **intr_priority**

MCPWM capture interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **prescale**

Prescale of input signal, effective frequency = cap_input_clk/prescale

uint32_t **pos_edge**

Whether to capture on positive edge

uint32_t **neg_edge**

Whether to capture on negative edge

uint32_t **pull_up**

Whether to pull up internally

uint32_t **pull_down**

Whether to pull down internally

uint32_t **invert_cap_signal**

Invert the input capture signal

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

uint32_t **keep_io_conf_at_exit**

For debug/test, whether to keep the GPIO configuration when capture channel is deleted. By default, driver will reset the GPIO pin at exit.

struct *mcpwm_capture_channel_config_t*::[anonymous] **flags**

Extra configuration flags for capture channel

struct **mcpwm_capture_event_callbacks_t**

Group of supported MCPWM capture event callbacks.

Note: The callbacks are all running under ISR environment

Public Members

mcpwm_capture_event_cb_t **on_cap**

Callback function that would be invoked when capture event occurred

Header File

- components/driver/include/driver/mcpwm_types.h

Structures

struct **mcpwm_timer_event_data_t**

MCPWM timer event data.

Public Members

uint32_t **count_value**

MCPWM timer count value

mcpwm_timer_direction_t **direction**

MCPWM timer count direction

struct **mcpwm_brake_event_data_t**

MCPWM brake event data.

struct **mcpwm_fault_event_data_t**

MCPWM fault event data.

struct **mcpwm_compare_event_data_t**

MCPWM compare event data.

Public Members

uint32_t **compare_ticks**

Compare value

mcpwm_timer_direction_t **direction**

Count direction

struct **mcpwm_capture_event_data_t**

MCPWM capture event data.

Public Members

uint32_t **cap_value**

Captured value

mcpwm_capture_edge_t **cap_edge**

Capture edge

Type Definitions

typedef struct mcpwm_timer_t ***mcpwm_timer_handle_t**

Type of MCPWM timer handle.

typedef struct mcpwm_oper_t ***mcpwm_oper_handle_t**

Type of MCPWM operator handle.

typedef struct mcpwm_cmpr_t ***mcpwm_cmpr_handle_t**

Type of MCPWM comparator handle.

typedef struct mcpwm_gen_t ***mcpwm_gen_handle_t**

Type of MCPWM generator handle.

typedef struct mcpwm_fault_t ***mcpwm_fault_handle_t**

Type of MCPWM fault handle.

typedef struct mcpwm_sync_t ***mcpwm_sync_handle_t**

Type of MCPWM sync handle.

typedef struct mcpwm_cap_timer_t ***mcpwm_cap_timer_handle_t**

Type of MCPWM capture timer handle.

typedef struct mcpwm_cap_channel_t ***mcpwm_cap_channel_handle_t**

Type of MCPWM capture channel handle.

typedef bool (***mcpwm_timer_event_cb_t**)(*mcpwm_timer_handle_t* timer, const *mcpwm_timer_event_data_t* *edata, void *user_ctx)

MCPWM timer event callback function.

Param timer **[in]** MCPWM timer handle
Param edata **[in]** MCPWM timer event data, fed by driver
Param user_ctx **[in]** User data, set in `mcpwm_timer_register_event_callbacks()`
Return Whether a high priority task has been waken up by this function

typedef bool (***mcpwm_brake_event_cb_t**)(*mcpwm_oper_handle_t* oper, const *mcpwm_brake_event_data_t* *edata, void *user_ctx)

MCPWM operator brake event callback function.

Param oper **[in]** MCPWM operator handle
Param edata **[in]** MCPWM brake event data, fed by driver
Param user_ctx **[in]** User data, set in `mcpwm_operator_register_event_callbacks()`
Return Whether a high priority task has been waken up by this function

typedef bool (***mcpwm_fault_event_cb_t**)(*mcpwm_fault_handle_t* fault, const *mcpwm_fault_event_data_t* *edata, void *user_ctx)

MCPWM fault event callback function.

Param fault MCPWM fault handle
Param edata MCPWM fault event data, fed by driver
Param user_ctx User data, set in `mcpwm_fault_register_event_callbacks()`
Return whether a task switch is needed after the callback returns

typedef bool (***mcpwm_compare_event_cb_t**)(*mcpwm_cmpr_handle_t* comparator, const *mcpwm_compare_event_data_t* *edata, void *user_ctx)

MCPWM comparator event callback function.

Param comparator MCPWM comparator handle
Param edata MCPWM comparator event data, fed by driver
Param user_ctx User data, set in `mcpwm_comparator_register_event_callbacks()`
Return Whether a high priority task has been waken up by this function

typedef bool (***mcpwm_capture_event_cb_t**)(*mcpwm_cap_channel_handle_t* cap_channel, const *mcpwm_capture_event_data_t* *edata, void *user_ctx)

MCPWM capture event callback function.

Param cap_channel MCPWM capture channel handle

Param edata MCPWM capture event data, fed by driver
Param user_ctx User data, set in `mcpwm_capture_channel_register_event_callbacks()`
Return Whether a high priority task has been waken up by this function

Header File

- components/hal/include/hal/mcpwm_types.h

Type Definitions

typedef *soc_periph_mcpwm_timer_clk_src_t* **mcpwm_timer_clock_source_t**

MCPWM timer clock source.

typedef *soc_periph_mcpwm_capture_clk_src_t* **mcpwm_capture_clock_source_t**

MCPWM capture clock source.

typedef *soc_periph_mcpwm_carrier_clk_src_t* **mcpwm_carrier_clock_source_t**

MCPWM carrier clock source.

Enumerations

enum **mcpwm_timer_direction_t**

MCPWM timer count direction.

Values:

enumerator **MCPWM_TIMER_DIRECTION_UP**

Counting direction: Increase

enumerator **MCPWM_TIMER_DIRECTION_DOWN**

Counting direction: Decrease

enum **mcpwm_timer_event_t**

MCPWM timer events.

Values:

enumerator **MCPWM_TIMER_EVENT_EMPTY**

MCPWM timer counts to zero (i.e. counter is empty)

enumerator **MCPWM_TIMER_EVENT_FULL**

MCPWM timer counts to peak (i.e. counter is full)

enumerator **MCPWM_TIMER_EVENT_INVALID**

MCPWM timer invalid event

enum **mcpwm_timer_count_mode_t**

MCPWM timer count modes.

Values:

enumerator **MCPWM_TIMER_COUNT_MODE_PAUSE**

MCPWM timer paused

enumerator **MCPWM_TIMER_COUNT_MODE_UP**

MCPWM timer counting up

enumerator **MCPWM_TIMER_COUNT_MODE_DOWN**

MCPWM timer counting down

enumerator **MCPWM_TIMER_COUNT_MODE_UP_DOWN**

MCPWM timer counting up and down

enum **mcpwm_timer_start_stop_cmd_t**

MCPWM timer commands, specify the way to start or stop the timer.

Values:

enumerator **MCPWM_TIMER_STOP_EMPTY**

MCPWM timer stops when next count reaches zero

enumerator **MCPWM_TIMER_STOP_FULL**

MCPWM timer stops when next count reaches peak

enumerator **MCPWM_TIMER_START_NO_STOP**

MCPWM timer starts couting, and don’ t stop until received stop command

enumerator **MCPWM_TIMER_START_STOP_EMPTY**

MCPWM timer starts counting and stops when next count reaches zero

enumerator **MCPWM_TIMER_START_STOP_FULL**

MCPWM timer starts counting and stops when next count reaches peak

enum **mcpwm_generator_action_t**

MCPWM generator actions.

Values:

enumerator **MCPWM_GEN_ACTION_KEEP**

Generator action: Keep the same level

enumerator **MCPWM_GEN_ACTION_LOW**

Generator action: Force to low level

enumerator **MCPWM_GEN_ACTION_HIGH**

Generator action: Force to high level

enumerator **MCPWM_GEN_ACTION_TOGGLE**

Generator action: Toggle level

enum **mcpwm_operator_brake_mode_t**

MCPWM operator brake mode.

Values:

enumerator **MCPWM_OPER_BRAKE_MODE_CBC**

Brake mode: CBC (cycle by cycle)

enumerator **MCPWM_OPER_BRAKE_MODE_OST**

Brake mode: OST (one shot)

enumerator **MCPWM_OPER_BRAKE_MODE_INVALID**

MCPWM operator invalid brake mode

enum **mcpwm_capture_edge_t**

MCPWM capture edge.

Values:

enumerator **MCPWM_CAP_EDGE_POS**

Capture on the positive edge

enumerator **MCPWM_CAP_EDGE_NEG**

Capture on the negative edge

2.6.13 Pulse Counter (PCNT)

Introduction

The PCNT (Pulse Counter) module is designed to count the number of rising and/or falling edges of input signals. The ESP32 contains multiple pulse counter units in the module.[1] Each unit is in effect an independent counter with multiple channels, where each channel can increment/decrement the counter on a rising/falling edge. Furthermore, each channel can be configured separately.

PCNT channels can react to signals of **edge** type and **level** type, however for simple applications, detecting the edge signal is usually sufficient. PCNT channels can be configured react to both pulse edges (i.e., rising and falling edge), and can be configured to increase, decrease or do nothing to the unit' s counter on each edge. The level signal is the so-called **control signal**, which is used to control the counting mode of the edge signals that are attached to the same channel. By combining the usage of both edge and level signals, a PCNT unit can act as a **quadrature decoder**.

Besides that, PCNT unit is equipped with a separate glitch filter, which is helpful to remove noise from the signal.

Typically, a PCNT module can be used in scenarios like:

- Calculate periodic signal' s frequency by counting the pulse numbers within a time slice
- Decode quadrature signals into speed and direction

Functional Overview

Description of the PCNT functionality is divided into the following sections:

- *Resource Allocation* - covers how to allocate PCNT units and channels with properly set of configurations. It also covers how to recycle the resources when they finished working.
- *Set Up Channel Actions* - covers how to configure the PCNT channel to behave on different signal edges and levels.
- *Watch Points* - describes how to configure PCNT watch points (i.e., tell PCNT unit to trigger an event when the count reaches a certain value).
- *Register Event Callbacks* - describes how to hook your specific code to the watch point event callback function.

[1] Different ESP chip series might have different number of PCNT units and channels. Please refer to the [TRM] for details. The driver won' t forbid you from applying for more PCNT units and channels, but it will return error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g. `pcnt_new_unit()`).

- *Set Glitch Filter* - describes how to enable and set the timing parameters for the internal glitch filter.
- *Enable and Disable Unit* - describes how to enable and disable the PCNT unit.
- *Unit IO Control* - describes IO control functions of PCNT unit, like enable glitch filter, start and stop unit, get and clear count value.
- *Power Management* - describes what functionality will prevent the chip from going into low power mode.
- *IRAM Safe* - describes tips on how to make the PCNT interrupt and IO control functions work better along with a disabled cache.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.
- *Kconfig Options* - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

Resource Allocation The PCNT unit and channel are represented by `pcnt_unit_handle_t` and `pcnt_channel_handle_t` respectively. All available units and channels are maintained by the driver in a resource pool, so you do not need to know the exact underlying instance ID.

Install PCNT Unit To install a PCNT unit, there's a configuration structure that needs to be given in advance: `pcnt_unit_config_t`:

- `pcnt_unit_config_t::low_limit` and `pcnt_unit_config_t::high_limit` specify the range for the internal hardware counter. The counter will reset to zero automatically when it crosses either the high or low limit.
- `pcnt_unit_config_t::accum_count` sets whether to create an internal accumulator for the counter. This is helpful when you want to extend the counter's width, which by default is 16bit at most, defined in the hardware. See also *Compensate Overflow Loss* for how to use this feature to compensate the overflow loss.
- `pcnt_unit_config_t::intr_priority` sets the priority of the timer interrupt. If it is set to `0`, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.

Note: Since all PCNT units share the same interrupt source, when installing multiple PCNT units make sure that the interrupt priority `pcnt_unit_config_t::intr_priority` is the same for each unit.

Unit allocation and initialization is done by calling a function `pcnt_new_unit()` with `pcnt_unit_config_t` as an input parameter. The function will return a PCNT unit handle only when it runs correctly. Specifically, when there are no more free PCNT units in the pool (i.e. unit resources have been used up), then this function will return `ESP_ERR_NOT_FOUND` error. The total number of available PCNT units is recorded by `SOC_PCNT_UNITS_PER_GROUP` for reference.

If a previously created PCNT unit is no longer needed, it's recommended to recycle the resource by calling `pcnt_del_unit()`. Which in return allows the underlying unit hardware to be used for other purposes. Before deleting a PCNT unit, one should ensure the following prerequisites:

- The unit is in the init state, in other words, the unit is either disabled by `pcnt_unit_disable()` or not enabled yet.
- The attached PCNT channels are all removed by `pcnt_del_channel()`.

```
#define EXAMPLE_PCNT_HIGH_LIMIT 100
#define EXAMPLE_PCNT_LOW_LIMIT  -100

pcnt_unit_config_t unit_config = {
    .high_limit = EXAMPLE_PCNT_HIGH_LIMIT,
    .low_limit = EXAMPLE_PCNT_LOW_LIMIT,
};
pcnt_unit_handle_t pcnt_unit = NULL;
ESP_ERROR_CHECK(pcnt_new_unit(&unit_config, &pcnt_unit));
```

Install PCNT Channel To install a PCNT channel, you must initialize a `pcnt_chan_config_t` structure in advance, and then call `pcnt_new_channel()`. The configuration fields of the `pcnt_chan_config_t` structure are described below:

- *pcnt_chan_config_t::edge_gpio_num* and *pcnt_chan_config_t::level_gpio_num* specify the GPIO numbers used by **edge** type signal and **level** type signal. Please note, either of them can be assigned to *-1* if it's not actually used, and thus it will become a **virtual IO**. For some simple pulse counting applications where one of the level/edge signals is fixed (i.e., never changes), you can reclaim a GPIO by setting the signal as a virtual IO on channel allocation. Setting the level/edge signal as a virtual IO will cause that signal to be internally routed to a fixed High/Low logic level, thus allowing you to save a GPIO for other purposes.
- *pcnt_chan_config_t::virt_edge_io_level* and *pcnt_chan_config_t::virt_level_io_level* specify the virtual IO level for **edge** and **level** input signal, to ensure a deterministic state for such control signal. Please note, they are only valid when either *pcnt_chan_config_t::edge_gpio_num* or *pcnt_chan_config_t::level_gpio_num* is assigned to *-1*.
- *pcnt_chan_config_t::invert_edge_input* and *pcnt_chan_config_t::invert_level_input* are used to decide whether to invert the input signals before they going into PCNT hardware. The invert is done by GPIO matrix instead of PCNT hardware.
- *pcnt_chan_config_t::io_loop_back* is for debug only, which enables both the GPIO's input and output paths. This can help to simulate the pulse signals by function *gpio_set_level()* on the same GPIO.

Channel allocating and initialization is done by calling a function *pcnt_new_channel()* with the above *pcnt_chan_config_t* as an input parameter plus a PCNT unit handle returned from *pcnt_new_unit()*. This function will return a PCNT channel handle if it runs correctly. Specifically, when there are no more free PCNT channel within the unit (i.e. channel resources have been used up), then this function will return *ESP_ERR_NOT_FOUND* error. The total number of available PCNT channels within the unit is recorded by *SOC_PCNT_CHANNELS_PER_UNIT* for reference. Note that, when install a PCNT channel for a specific unit, one should ensure the unit is in the init state, otherwise this function will return *ESP_ERR_INVALID_STATE* error.

If a previously created PCNT channel is no longer needed, it's recommended to recycle the resources by calling *pcnt_del_channel()*. Which in return allows the underlying channel hardware to be used for other purposes.

```
#define EXAMPLE_CHAN_GPIO_A 0
#define EXAMPLE_CHAN_GPIO_B 2

pcnt_chan_config_t chan_config = {
    .edge_gpio_num = EXAMPLE_CHAN_GPIO_A,
    .level_gpio_num = EXAMPLE_CHAN_GPIO_B,
};
pcnt_channel_handle_t pcnt_chan = NULL;
ESP_ERROR_CHECK(pcnt_new_channel(pcnt_unit, &chan_config, &pcnt_chan));
```

Set Up Channel Actions The PCNT will increase/decrease/hold its internal count value when the input pulse signal toggles. You can set different actions for edge signal and/or level signal.

- *pcnt_channel_set_edge_action()* function is to set specific actions for rising and falling edge of the signal attached to the *pcnt_chan_config_t::edge_gpio_num*. Supported actions are listed in *pcnt_channel_edge_action_t*.
- *pcnt_channel_set_level_action()* function is to set specific actions for high and low level of the signal attached to the *pcnt_chan_config_t::level_gpio_num*. Supported actions are listed in *pcnt_channel_level_action_t*. This function is not mandatory if the *pcnt_chan_config_t::level_gpio_num* is set to *-1* when allocating PCNT channel by *pcnt_new_channel()*.

```
// decrease the counter on rising edge, increase the counter on falling edge
ESP_ERROR_CHECK(pcnt_channel_set_edge_action(pcnt_chan, PCNT_CHANNEL_EDGE_ACTION_
↪DECREASE, PCNT_CHANNEL_EDGE_ACTION_INCREASE));
// keep the counting mode when the control signal is high level, and reverse the␣
↪counting mode when the control signal is low level
ESP_ERROR_CHECK(pcnt_channel_set_level_action(pcnt_chan, PCNT_CHANNEL_LEVEL_ACTION_
↪KEEP, PCNT_CHANNEL_LEVEL_ACTION_INVERSE));
```

Watch Points Each PCNT unit can be configured to watch several different values that you're interested in. The value to be watched is also called **Watch Point**. The watch point itself can't exceed the range set in *pcnt_unit_config_t* by *pcnt_unit_config_t::low_limit* and *pcnt_unit_config_t::high_limit*. When the counter reaches either watch point, a watch event will be triggered and notify you by interrupt if any watch event callback has ever registered in *pcnt_unit_register_event_callbacks()*. See *Register Event Callbacks* for how to register event callbacks.

The watch point can be added and removed by *pcnt_unit_add_watch_point()* and *pcnt_unit_remove_watch_point()*. The commonly used watch points are: **zero cross**, **maximum / minimum count** and other threshold values. The number of available watch point is limited, *pcnt_unit_add_watch_point()* will return error *ESP_ERR_NOT_FOUND* if it can't find any free hardware resource to save the watch point. You can't add the same watch point for multiple times, otherwise it will return error *ESP_ERR_INVALID_STATE*.

It is recommended to remove the unused watch point by *pcnt_unit_remove_watch_point()* to recycle the watch point resources.

```
// add zero across watch point
ESP_ERROR_CHECK(pcnt_unit_add_watch_point(pcnt_unit, 0));
// add high limit watch point
ESP_ERROR_CHECK(pcnt_unit_add_watch_point(pcnt_unit, EXAMPLE_PCNT_HIGH_LIMIT));
```

Register Event Callbacks When PCNT unit reaches any enabled watch point, specific event will be generated and notify the CPU by interrupt. If you have some function that want to get executed when event happens, you should hook your function to the interrupt service routine by calling *pcnt_unit_register_event_callbacks()*. All supported event callbacks are listed in the *pcnt_event_callbacks_t*:

- *pcnt_event_callbacks_t::on_reach* sets a callback function for watch point event. As this function is called within the ISR context, you must ensure that the function doesn't attempt to block (e.g., by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The function prototype is declared in *pcnt_watch_cb_t*.

You can save their own context to *pcnt_unit_register_event_callbacks()* as well, via the parameter `user_ctx`. This user data will be directly passed to the callback functions.

In the callback function, the driver will fill in the event data of specific event. For example, the watch point event data is declared as *pcnt_watch_event_data_t*:

- *pcnt_watch_event_data_t::watch_point_value* saves the watch point value that triggers the event.
- *pcnt_watch_event_data_t::zero_cross_mode* saves how the PCNT unit crosses the zero point in the latest time. The possible zero cross modes are listed in the *pcnt_unit_zero_cross_mode_t*. Usually different zero cross mode means different **counting direction** and **counting step size**.

Registering callback function will result in lazy installation of interrupt service, thus this function should only be called before the unit is enabled by *pcnt_unit_enable()*. Otherwise, it can return *ESP_ERR_INVALID_STATE* error.

```
static bool example_pcnt_on_reach(pcnt_unit_handle_t unit, const pcnt_watch_event_
↪data_t *edata, void *user_ctx)
{
    BaseType_t high_task_wakeup;
    QueueHandle_t queue = (QueueHandle_t)user_ctx;
    // send watch point to queue, from this interrupt callback
    xQueueSendFromISR(queue, &(edata->watch_point_value), &high_task_wakeup);
    // return whether a high priority task has been waken up by this function
    return (high_task_wakeup == pdTRUE);
}

pcnt_event_callbacks_t cbs = {
```

(continues on next page)

(continued from previous page)

```
    .on_reach = example_pcnt_on_reach,
};
QueueHandle_t queue = xQueueCreate(10, sizeof(int));
ESP_ERROR_CHECK(pcnt_unit_register_event_callbacks(pcnt_unit, &cbs, queue));
```

Set Glitch Filter The PCNT unit features filters to ignore possible short glitches in the signals. The parameters that can be configured for the glitch filter are listed in *pcnt_glitch_filter_config_t*:

- *pcnt_glitch_filter_config_t::max_glitch_ns* sets the maximum glitch width, in nano seconds. If a signal pulse's width is smaller than this value, then it will be treated as noise and won't increase/decrease the internal counter.

You can enable the glitch filter for PCNT unit by calling *pcnt_unit_set_glitch_filter()* with the filter configuration provided above. Particularly, you can disable the glitch filter later by calling *pcnt_unit_set_glitch_filter()* with a *NULL* filter configuration.

This function should be called when the the unit is in the init state. Otherwise, it will return *ESP_ERR_INVALID_STATE* error.

Note: The glitch filter is clocked from APB. For the counter not to miss any pulses, the maximum glitch width should be longer than one APB_CLK cycle (usually 12.5 ns if APB equals 80MHz). As the APB frequency would be changed after DFS (Dynamic Frequency Scaling) enabled, which means the filter won't work as expect in that case. So the driver will install a PM lock for PCNT unit during the first time you enable the glitch filter. For more information related to power management strategy used in PCNT driver, please see *Power Management*.

```
pcnt_glitch_filter_config_t filter_config = {
    .max_glitch_ns = 1000,
};
ESP_ERROR_CHECK(pcnt_unit_set_glitch_filter(pcnt_unit, &filter_config));
```

Enable and Disable Unit Before doing IO control to the PCNT unit, you need to enable it first, by calling *pcnt_unit_enable()*. Internally, this function will:

- switch the PCNT driver state from **init** to **enable**.
- enable the interrupt service if it has been lazy installed in *pcnt_unit_register_event_callbacks()*.
- acquire a proper power management lock if it has been lazy installed in *pcnt_unit_set_glitch_filter()*. See also *Power Management* for more information.

On the contrary, calling *pcnt_unit_disable()* will do the opposite, that is, put the PCNT driver back to the **init** state, disable the interrupts service and release the power management lock.

Unit IO Control

Start/Stop and Clear Calling *pcnt_unit_start()* will make the PCNT unit start to work, increase or decrease counter according to pulse signals. On the contrary, calling *pcnt_unit_stop()* will stop the PCNT unit but retain current count value. Instead, clearing counter can only be done by calling *pcnt_unit_clear_count()*.

Note, *pcnt_unit_start()* and *pcnt_unit_stop()* should be called when the unit has been enabled by *pcnt_unit_enable()*. Otherwise, it will return *ESP_ERR_INVALID_STATE* error.

Get Count Value You can read current count value at any time by calling *pcnt_unit_get_count()*. The returned count value is a **signed** integer, where the sign can be used to reflect the direction.

```
int pulse_count = 0;
ESP_ERROR_CHECK(pcnt_unit_get_count(pcnt_unit, &pulse_count));
```

Compensate Overflow Loss The internal hardware counter will be cleared to zero automatically when it reaches high or low limit. If you want to compensate for that count loss and extend the counter's bit-width, you can:

1. Enable *pcnt_unit_config_t::accum_count* when installing the PCNT unit.
2. Add the high/low limit as the *Watch Points*.
3. Now, the returned count value from the *pcnt_unit_get_count()* function not only reflects the hardware's count value, but also accumulates the high/low overflow loss to it.

Note: *pcnt_unit_clear_count()* will reset the accumulated count value as well.

Power Management When power management is enabled (i.e. *CONFIG_PM_ENABLE* is on), the system will adjust the APB frequency before going into light sleep, thus potentially changing the behavior of PCNT glitch filter and leading to valid signal being treated as noise.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type *ESP_PM_APB_FREQ_MAX*. Whenever you enable the glitch filter by *pcnt_unit_set_glitch_filter()*, the driver will guarantee that the power management lock is acquired after the PCNT unit is enabled by *pcnt_unit_enable()*. Likewise, the driver releases the lock after *pcnt_unit_disable()* is called.

IRAM Safe By default, the PCNT interrupt will be deferred when the Cache is disabled for reasons like writing/erasing Flash. Thus the alarm interrupt will not get executed in time, which is not expected in a real-time application.

There's a Kconfig option *CONFIG_PCNT_ISR_IRAM_SAFE* that will:

1. Enable the interrupt being serviced even when cache is disabled
2. Place all functions that used by the ISR into IRAM[2]
3. Place driver object into DRAM (in case it's mapped to PSRAM by accident)

This will allow the interrupt to run while the cache is disabled but will come at the cost of increased IRAM consumption.

There's another Kconfig option *CONFIG_PCNT_CTRL_FUNC_IN_IRAM* that can put commonly used IO control functions into IRAM as well. So that these functions can also be executable when the cache is disabled. These IO control functions are as follows:

- *pcnt_unit_start()*
- *pcnt_unit_stop()*
- *pcnt_unit_clear_count()*
- *pcnt_unit_get_count()*

Thread Safety The factory functions *pcnt_new_unit()* and *pcnt_new_channel()* are guaranteed to be thread safe by the driver, which means, you can call them from different RTOS tasks without protection by extra locks. The following functions are allowed to run under ISR context, the driver uses a critical section to prevent them being called concurrently in both task and ISR.

- *pcnt_unit_start()*
- *pcnt_unit_stop()*
- *pcnt_unit_clear_count()*
- *pcnt_unit_get_count()*

[2] *pcnt_event_callbacks_t::on_reach* callback and the functions invoked by itself should also be placed in IRAM, you need to take care of them by themselves.

Other functions that take the *pcnt_unit_handle_t* and *pcnt_channel_handle_t* as the first positional parameter, are not treated as thread safe. This means you should avoid calling them from multiple tasks.

Kconfig Options

- *CONFIG_PCNT_CTRL_FUNC_IN_IRAM* controls where to place the PCNT control functions (IRAM or Flash), see *IRAM Safe* for more information.
- *CONFIG_PCNT_ISR_IRAM_SAFE* controls whether the default ISR handler can work when cache is disabled, see *IRAM Safe* for more information.
- *CONFIG_PCNT_ENABLE_DEBUG_LOG* is used to enabled the debug log output. Enable this option will increase the firmware binary size.

Application Examples

- Decode the quadrature signals from rotary encoder: peripherals/pcnt/rotary_encoder.

API Reference

Header File

- components/driver/include/driver/pulse_cnt.h

Functions

esp_err_t **pcnt_new_unit**(const *pcnt_unit_config_t* *config, *pcnt_unit_handle_t* *ret_unit)

Create a new PCNT unit, and return the handle.

Note: The newly created PCNT unit is put in the init state.

Parameters

- **config** –**[in]** PCNT unit configuration
- **ret_unit** –**[out]** Returned PCNT unit handle

Returns

- ESP_OK: Create PCNT unit successfully
- ESP_ERR_INVALID_ARG: Create PCNT unit failed because of invalid argument (e.g. high/low limit value out of the range)
- ESP_ERR_NO_MEM: Create PCNT unit failed because out of memory
- ESP_ERR_NOT_FOUND: Create PCNT unit failed because all PCNT units are used up and no more free one
- ESP_FAIL: Create PCNT unit failed because of other error

esp_err_t **pcnt_del_unit**(*pcnt_unit_handle_t* unit)

Delete the PCNT unit handle.

Note: A PCNT unit can't be in the enable state when this function is invoked. See also `pcnt_unit_disable()` for how to disable a unit.

Parameters **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`

Returns

- ESP_OK: Delete the PCNT unit successfully
- ESP_ERR_INVALID_ARG: Delete the PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete the PCNT unit failed because the unit is not in init state or some PCNT channel is still in working
- ESP_FAIL: Delete the PCNT unit failed because of other error

esp_err_t **pcnt_unit_set_glitch_filter** (*pcnt_unit_handle_t* unit, const *pcnt_glitch_filter_config_t* *config)

Set glitch filter for PCNT unit.

Note: The glitch filter module is clocked from APB, and APB frequency can be changed during DFS, which in return make the filter out of action. So this function will lazy-install a PM lock internally when the power management is enabled. With this lock, the APB frequency won't be changed. The PM lock can be uninstalled in `pcnt_del_unit()`.

Note: This function should be called when the PCNT unit is in the init state (i.e. before calling `pcnt_unit_enable()`)

Parameters
- **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **config** –**[in]** PCNT filter configuration, set config to NULL means disabling the filter function

Returns
- ESP_OK: Set glitch filter successfully
- ESP_ERR_INVALID_ARG: Set glitch filter failed because of invalid argument (e.g. glitch width is too big)
- ESP_ERR_INVALID_STATE: Set glitch filter failed because the unit is not in the init state
- ESP_FAIL: Set glitch filter failed because of other error

esp_err_t **pcnt_unit_enable** (*pcnt_unit_handle_t* unit)

Enable the PCNT unit.

Note: This function will transit the unit state from init to enable.

Note: This function will enable the interrupt service, if it's lazy installed in `pcnt_unit_register_event_callbacks()`.

Note: This function will acquire the PM lock if it's lazy installed in `pcnt_unit_set_glitch_filter()`.

Note: Enable a PCNT unit doesn't mean to start it. See also `pcnt_unit_start()` for how to start the PCNT counter.

Parameters **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`

Returns
- ESP_OK: Enable PCNT unit successfully
- ESP_ERR_INVALID_ARG: Enable PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable PCNT unit failed because the unit is already enabled
- ESP_FAIL: Enable PCNT unit failed because of other error

esp_err_t **pcnt_unit_disable** (*pcnt_unit_handle_t* unit)

Disable the PCNT unit.

Note: This function will do the opposite work to the `pcnt_unit_enable()`

Note: Disable a PCNT unit doesn't mean to stop it. See also `pcnt_unit_stop()` for how to stop the PCNT counter.

Parameters **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
Returns
- ESP_OK: Disable PCNT unit successfully
- ESP_ERR_INVALID_ARG: Disable PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable PCNT unit failed because the unit is not enabled yet
- ESP_FAIL: Disable PCNT unit failed because of other error

esp_err_t **`pcnt_unit_start`**(*pcnt_unit_handle_t* unit)

Start the PCNT unit, the counter will start to count according to the edge and/or level input signals.

Note: This function should be called when the unit is in the enable state (i.e. after calling `pcnt_unit_enable()`)

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if `CONFIG_PCNT_CTRL_FUNC_IN_IRAM` is on, so that it's allowed to be executed when Cache is disabled

Parameters **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
Returns
- ESP_OK: Start PCNT unit successfully
- ESP_ERR_INVALID_ARG: Start PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Start PCNT unit failed because the unit is not enabled yet
- ESP_FAIL: Start PCNT unit failed because of other error

esp_err_t **`pcnt_unit_stop`**(*pcnt_unit_handle_t* unit)

Stop PCNT from counting.

Note: This function should be called when the unit is in the enable state (i.e. after calling `pcnt_unit_enable()`)

Note: The stop operation won't clear the counter. Also see `pcnt_unit_clear_count()` for how to clear pulse count value.

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if `CONFIG_PCNT_CTRL_FUNC_IN_IRAM`, so that it is allowed to be executed when Cache is disabled

Parameters **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
Returns
- ESP_OK: Stop PCNT unit successfully
- ESP_ERR_INVALID_ARG: Stop PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Stop PCNT unit failed because the unit is not enabled yet
- ESP_FAIL: Stop PCNT unit failed because of other error

esp_err_t **pcnt_unit_clear_count**(*pcnt_unit_handle_t* unit)

Clear PCNT pulse count value to zero.

Note: It's recommended to call this function after adding a watch point by `pcnt_unit_add_watch_point()`, so that the newly added watch point is effective immediately.

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if `CONFIG_PCNT_CTRL_FUNC_IN_IRAM`, so that it's allowed to be executed when Cache is disabled

Parameters **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
Returns
- ESP_OK: Clear PCNT pulse count successfully
- ESP_ERR_INVALID_ARG: Clear PCNT pulse count failed because of invalid argument
- ESP_FAIL: Clear PCNT pulse count failed because of other error

esp_err_t **pcnt_unit_get_count**(*pcnt_unit_handle_t* unit, int *value)

Get PCNT count value.

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if `CONFIG_PCNT_CTRL_FUNC_IN_IRAM`, so that it's allowed to be executed when Cache is disabled

Parameters
- **unit** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **value** –**[out]** Returned count value

Returns
- ESP_OK: Get PCNT pulse count successfully
- ESP_ERR_INVALID_ARG: Get PCNT pulse count failed because of invalid argument
- ESP_FAIL: Get PCNT pulse count failed because of other error

esp_err_t **pcnt_unit_register_event_callbacks**(*pcnt_unit_handle_t* unit, const *pcnt_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for PCNT unit.

Note: User registered callbacks are expected to be runnable within ISR context

Note: The first call to this function needs to be before the call to `pcnt_unit_enable`

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Parameters

- **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **`cbs`** –**[in]** Group of callback functions
- **`user_data`** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because the unit is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **`pcnt_unit_add_watch_point`**(*pcnt_unit_handle_t* unit, int watch_point)

Add a watch point for PCNT unit, PCNT will generate an event when the counter value reaches the watch point value.

Parameters

- **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **`watch_point`** –**[in]** Value to be watched

Returns

- ESP_OK: Add watch point successfully
- ESP_ERR_INVALID_ARG: Add watch point failed because of invalid argument (e.g. the value to be watched is out of the limitation set in *`pcnt_unit_config_t`*)
- ESP_ERR_INVALID_STATE: Add watch point failed because the same watch point has already been added
- ESP_ERR_NOT_FOUND: Add watch point failed because no more hardware watch point can be configured
- ESP_FAIL: Add watch point failed because of other error

esp_err_t **`pcnt_unit_remove_watch_point`**(*pcnt_unit_handle_t* unit, int watch_point)

Remove a watch point for PCNT unit.

Parameters

- **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **`watch_point`** –**[in]** Watch point value

Returns

- ESP_OK: Remove watch point successfully
- ESP_ERR_INVALID_ARG: Remove watch point failed because of invalid argument
- ESP_ERR_INVALID_STATE: Remove watch point failed because the watch point was not added by `pcnt_unit_add_watch_point()` yet
- ESP_FAIL: Remove watch point failed because of other error

esp_err_t **`pcnt_new_channel`**(*pcnt_unit_handle_t* unit, const *pcnt_chan_config_t* *config, *pcnt_channel_handle_t* *ret_chan)

Create PCNT channel for specific unit, each PCNT has several channels associated with it.

Note: This function should be called when the unit is in init state (i.e. before calling `pcnt_unit_enable()`)

Parameters

- **`unit`** –**[in]** PCNT unit handle created by `pcnt_new_unit()`
- **`config`** –**[in]** PCNT channel configuration
- **`ret_chan`** –**[out]** Returned channel handle

Returns

- ESP_OK: Create PCNT channel successfully
- ESP_ERR_INVALID_ARG: Create PCNT channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create PCNT channel failed because of insufficient memory
- ESP_ERR_NOT_FOUND: Create PCNT channel failed because all PCNT channels are used up and no more free one
- ESP_ERR_INVALID_STATE: Create PCNT channel failed because the unit is not in the init state
- ESP_FAIL: Create PCNT channel failed because of other error

esp_err_t **pcnt_del_channel** (*pcnt_channel_handle_t* chan)

Delete the PCNT channel.

Parameters **chan** –**[in]** PCNT channel handle created by `pcnt_new_channel()`

Returns

- ESP_OK: Delete the PCNT channel successfully
- ESP_ERR_INVALID_ARG: Delete the PCNT channel failed because of invalid argument
- ESP_FAIL: Delete the PCNT channel failed because of other error

esp_err_t **pcnt_channel_set_edge_action** (*pcnt_channel_handle_t* chan, *pcnt_channel_edge_action_t* pos_act, *pcnt_channel_edge_action_t* neg_act)

Set channel actions when edge signal changes (e.g. falling or rising edge occurred). The edge signal is input from the `edge_gpio_num` configured in *`pcnt_chan_config_t`*. We use these actions to control when and how to change the counter value.

Parameters

- **chan** –**[in]** PCNT channel handle created by `pcnt_new_channel()`
- **pos_act** –**[in]** Action on posedge signal
- **neg_act** –**[in]** Action on negedge signal

Returns

- ESP_OK: Set edge action for PCNT channel successfully
- ESP_ERR_INVALID_ARG: Set edge action for PCNT channel failed because of invalid argument
- ESP_FAIL: Set edge action for PCNT channel failed because of other error

esp_err_t **pcnt_channel_set_level_action** (*pcnt_channel_handle_t* chan, *pcnt_channel_level_action_t* high_act, *pcnt_channel_level_action_t* low_act)

Set channel actions when level signal changes (e.g. signal level goes from high to low). The level signal is input from the `level_gpio_num` configured in *`pcnt_chan_config_t`*. We use these actions to control when and how to change the counting mode.

Parameters

- **chan** –**[in]** PCNT channel handle created by `pcnt_new_channel()`
- **high_act** –**[in]** Action on high level signal
- **low_act** –**[in]** Action on low level signal

Returns

- ESP_OK: Set level action for PCNT channel successfully
- ESP_ERR_INVALID_ARG: Set level action for PCNT channel failed because of invalid argument
- ESP_FAIL: Set level action for PCNT channel failed because of other error

Structures

struct **pcnt_watch_event_data_t**

PCNT watch event data.

Public Members

int **watch_point_value**

Watch point value that triggered the event

pcnt_unit_zero_cross_mode_t **zero_cross_mode**

Zero cross mode

struct **pcnt_event_callbacks_t**

Group of supported PCNT callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_PCNT_ISR_IRAM_SAFE is enabled, the callback itself and functions callbed by it should be placed in IRAM.

Public Members

pcnt_watch_cb_t **on_reach**

Called when PCNT unit counter reaches any watch point

struct **pcnt_unit_config_t**

PCNT unit configuration.

Public Members

int **low_limit**

Low limitation of the count unit, should be lower than 0

int **high_limit**

High limitation of the count unit, should be higher than 0

int **intr_priority**

PCNT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **accum_count**

Whether to accumulate the count value when overflows at the high/low limit

struct *pcnt_unit_config_t*::[anonymous] **flags**

Extra flags

struct **pcnt_chan_config_t**

PCNT channel configuration.

Public Members

int **edge_gpio_num**

GPIO number used by the edge signal, input mode with pull up enabled. Set to -1 if unused

int **level_gpio_num**

GPIO number used by the level signal, input mode with pull up enabled. Set to -1 if unused

uint32_t **invert_edge_input**

Invert the input edge signal

uint32_t **invert_level_input**

Invert the input level signal

uint32_t **virt_edge_io_level**

Virtual edge IO level, 0: low, 1: high. Only valid when edge_gpio_num is set to -1

uint32_t **virt_level_io_level**

Virtual level IO level, 0: low, 1: high. Only valid when level_gpio_num is set to -1

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

struct *pcnt_chan_config_t*::[anonymous] **flags**

Channel config flags

struct **pcnt_glitch_filter_config_t**

PCNT glitch filter configuration.

Public Members

uint32_t **max_glitch_ns**

Pulse width smaller than this threshold will be treated as glitch and ignored, in the unit of ns

Type Definitions

typedef struct pcnt_unit_t ***pcnt_unit_handle_t**

Type of PCNT unit handle.

typedef struct pcnt_chan_t ***pcnt_channel_handle_t**

Type of PCNT channel handle.

typedef bool (***pcnt_watch_cb_t**)(*pcnt_unit_handle_t* unit, const *pcnt_watch_event_data_t* *edata, void *user_ctx)

PCNT watch event callback prototype.

Note: The callback function is invoked from an ISR context, so it should meet the restrictions of not calling any blocking APIs when implementing the callback. e.g. must use ISR version of FreeRTOS APIs.

Param unit **[in]** PCNT unit handle

Param edata [in] PCNT event data, fed by the driver
Param user_ctx [in] User data, passed from `pcnt_unit_register_event_callbacks()`
Return Whether a high priority task has been woken up by this function

Header File

- components/hal/include/hal/pcnt_types.h

Enumerations

enum **pcnt_channel_level_action_t**

PCNT channel action on control level.

Values:

enumerator **PCNT_CHANNEL_LEVEL_ACTION_KEEP**

Keep current count mode

enumerator **PCNT_CHANNEL_LEVEL_ACTION_INVERSE**

Invert current count mode (increase -> decrease, decrease -> increase)

enumerator **PCNT_CHANNEL_LEVEL_ACTION_HOLD**

Hold current count value

enum **pcnt_channel_edge_action_t**

PCNT channel action on signal edge.

Values:

enumerator **PCNT_CHANNEL_EDGE_ACTION_HOLD**

Hold current count value

enumerator **PCNT_CHANNEL_EDGE_ACTION_INCREASE**

Increase count value

enumerator **PCNT_CHANNEL_EDGE_ACTION_DECREASE**

Decrease count value

enum **pcnt_unit_zero_cross_mode_t**

PCNT unit zero cross mode.

Values:

enumerator **PCNT_UNIT_ZERO_CROSS_POS_ZERO**

start from positive value, end to zero, i.e. +N->0

enumerator **PCNT_UNIT_ZERO_CROSS_NEG_ZERO**

start from negative value, end to zero, i.e. -N->0

enumerator **PCNT_UNIT_ZERO_CROSS_NEG_POS**

start from negative value, end to positive value, i.e. -N->+M

enumerator **PCNT_UNIT_ZERO_CROSS_POS_NEG**

start from positive value, end to negative value, i.e. +N->-M

2.6.14 Remote Control Transceiver (RMT)

Introduction

The RMT (Remote Control Transceiver) peripheral was designed to act as an infrared transceiver. However, due to the flexibility of its data format, RMT can be extended to a versatile and general-purpose transceiver, transmitting or receiving many other types of signals. From the perspective of network layering, the RMT hardware contains both physical and data link layers. The physical layer defines the communication media and bit signal representation. The data link layer defines the format of an RMT frame. The minimal data unit in the frame is called the **RMT symbol**, which is represented by `rmt_symbol_word_t` in the driver.

ESP32 contains multiple channels in the RMT peripheral[1]. Each channel can be independently configured as either transmitter or receiver.

Typically, the RMT peripheral can be used in the following scenarios:

- Transmit or receive infrared signals, with any IR protocols, e.g., NEC
- General-purpose sequence generator
- Transmit signals in a hardware-controlled loop, with a finite or infinite number of times
- Multi-channel simultaneous transmission
- Modulate the carrier to the output signal or demodulate the carrier from the input signal

Layout of RMT Symbols The RMT hardware defines data in its own pattern –the **RMT symbol**. The diagram below illustrates the bit fields of an RMT symbol. Each symbol consists of two pairs of two values. The first value in the pair is a 15-bit value representing the signal's duration in units of RMT ticks. The second in the pair is a 1-bit value representing the signal's logic level, i.e., high or low.

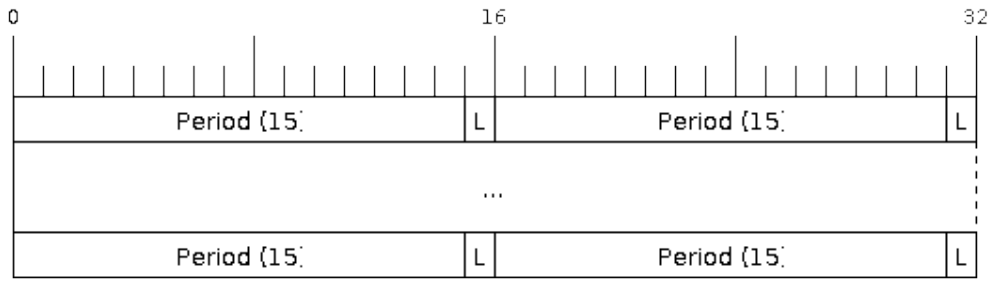


Fig. 15: Structure of RMT symbols (L - signal level)

RMT Transmitter Overview The data path and control path of an RMT TX channel is illustrated in the figure below:

The driver encodes the user's data into RMT data format, then the RMT transmitter can generate the waveforms according to the encoding artifacts. It is also possible to modulate a high-frequency carrier signal before being routed to a GPIO pad.

RMT Receiver Overview The data path and control path of an RMT RX channel is illustrated in the figure below:

[1] Different ESP chip series might have different numbers of RMT channels. Please refer to [TRM] for details. The driver does not forbid you from applying for more RMT channels, but it returns an error when there are no hardware resources available. Please always check the return value when doing *Resource Allocation*.

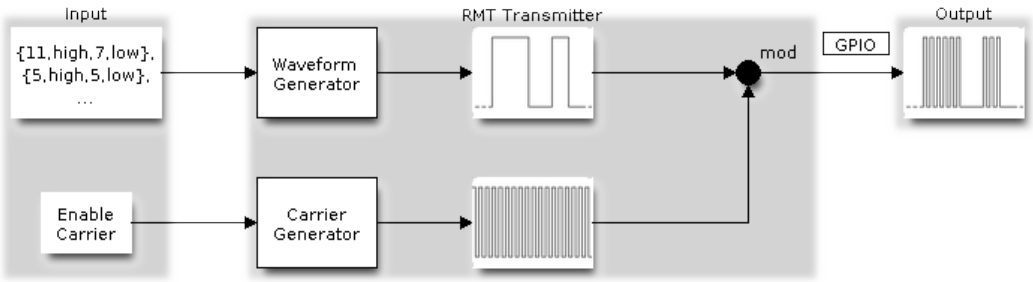


Fig. 16: RMT Transmitter Overview

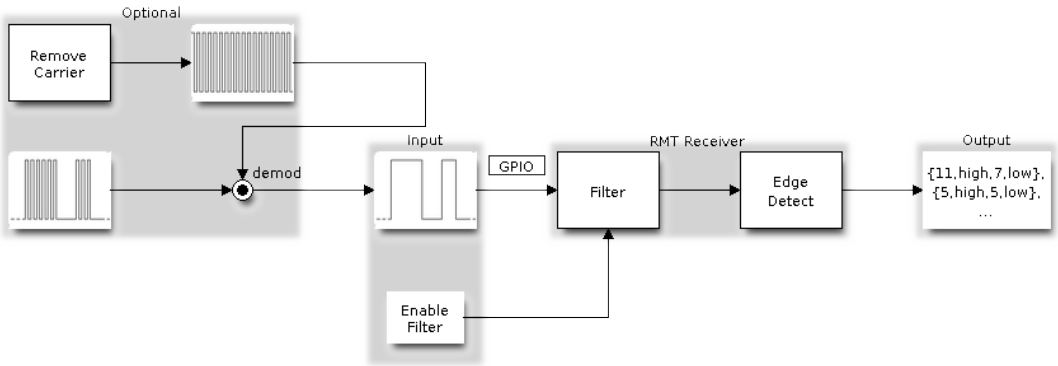


Fig. 17: RMT Receiver Overview

The RMT receiver can sample incoming signals into RMT data format, and store the data in memory. It is also possible to tell the receiver the basic characteristics of the incoming signal, so that the signal's stop condition can be recognized, and signal glitches and noise can be filtered out. The RMT peripheral also supports demodulating the high-frequency carrier from the base signal.

Functional Overview

The description of the RMT functionality is divided into the following sections:

- *Resource Allocation* - covers how to allocate and properly configure RMT channels. It also covers how to recycle channels and other resources when they are no longer used.
- *Carrier Modulation and Demodulation* - describes how to modulate and demodulate the carrier signals for TX and RX channels respectively.
- *Register Event Callbacks* - covers how to register user-provided event callbacks to receive RMT channel events.
- *Enable and Disable Channel* - shows how to enable and disable the RMT channel.
- *Initiate TX Transaction* - describes the steps to initiate a transaction for a TX channel.
- *Initiate RX Transaction* - describes the steps to initiate a transaction for an RX channel.
- *Multiple Channels Simultaneous Transmission* - describes how to collect multiple channels into a sync group so that their transmissions can be started simultaneously.
- *RMT Encoder* - focuses on how to write a customized encoder by combining multiple primitive encoders that are provided by the driver.
- *Power Management* - describes how different clock sources affects power consumption.
- *IRAM Safe* - describes how disabling the cache affects the RMT driver, and tips to mitigate it.
- *Thread Safety* - lists which APIs are guaranteed to be thread-safe by the driver.
- *Kconfig Options* - describes the various Kconfig options supported by the RMT driver.

Resource Allocation Both RMT TX and RX channels are represented by `rmt_channel_handle_t` in the driver. The driver internally manages which channels are available and hands out a free channel on request.

Install RMT TX Channel To install an RMT TX channel, there is a configuration structure that needs to be given in advance `rmt_tx_channel_config_t`. The following list describes each member of the configuration structure.

- `rmt_tx_channel_config_t::gpio_num` sets the GPIO number used by the transmitter.
- `rmt_tx_channel_config_t::clk_src` selects the source clock for the RMT channel. The available clocks are listed in `rmt_clock_source_t`. Note that, the selected clock is also used by other channels, which means the user should ensure this configuration is the same when allocating other channels, regardless of TX or RX. For the effect on the power consumption of different clock sources, please refer to the *Power Management* section.
- `rmt_tx_channel_config_t::resolution_hz` sets the resolution of the internal tick counter. The timing parameter of the RMT signal is calculated based on this **tick**.
- `rmt_tx_channel_config_t::mem_block_symbols` has a slightly different meaning based on if the DMA backend is enabled or not.
  - If the DMA is enabled via `rmt_tx_channel_config_t::with_dma`, then this field controls the size of the internal DMA buffer. To achieve a better throughput and smaller CPU overhead, you can set a larger value, e.g., `1024`.
  - If DMA is not used, this field controls the size of the dedicated memory block owned by the channel, which should be at least 64.
- `rmt_tx_channel_config_t::trans_queue_depth` sets the depth of the internal transaction queue, the deeper the queue, the more transactions can be prepared in the backlog.
- `rmt_tx_channel_config_t::invert_out` is used to decide whether to invert the RMT signal before sending it to the GPIO pad.
- `rmt_tx_channel_config_t::with_dma` enables the DMA backend for the channel. Using the DMA allows a significant amount of the channel's workload to be offloaded from the CPU. However, the DMA backend is not available on all ESP chips, please refer to [TRM] before you enable this option. Or you might encounter a `ESP_ERR_NOT_SUPPORTED` error.

- *rmt_tx_channel_config_t::io_loop_back* enables both input and output capabilities on the channel's assigned GPIO. Thus, by binding a TX and RX channel to the same GPIO, loopback can be achieved.
- *rmt_tx_channel_config_t::io_od_mode* configures the channel's assigned GPIO as open-drain. When combined with *rmt_tx_channel_config_t::io_loop_back*, a bi-directional bus (e.g., 1-wire) can be achieved.
- *rmt_tx_channel_config_t::intr_priority* Set the priority of the interrupt. If set to `0` , then the driver will use a interrupt with low or medium priority (priority level may be one of 1,2 or 3), otherwise use the priority indicated by *rmt_tx_channel_config_t::intr_priority*. Please use the number form (1,2,3) , not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until *rmt_del_channel()* is called.

Once the *rmt_tx_channel_config_t* structure is populated with mandatory parameters, users can call *rmt_new_tx_channel()* to allocate and initialize a TX channel. This function returns an RMT channel handle if it runs correctly. Specifically, when there are no more free channels in the RMT resource pool, this function returns *ESP_ERR_NOT_FOUND* error. If some feature (e.g., DMA backend) is not supported by the hardware, it returns *ESP_ERR_NOT_SUPPORTED* error.

```
rmt_channel_handle_t tx_chan = NULL;
rmt_tx_channel_config_t tx_chan_config = {
    .clk_src = RMT_CLK_SRC_DEFAULT,   // select source clock
    .gpio_num = 0,                    // GPIO number
    .mem_block_symbols = 64,          // memory block size, 64 * 4 = 256 Bytes
    .resolution_hz = 1 * 1000 * 1000, // 1 MHz tick resolution, i.e., 1 tick = 1 µs
    .trans_queue_depth = 4,           // set the number of transactions that can␣
↪pend in the background
    .flags.invert_out = false,        // do not invert output signal
    .flags.with_dma = false,          // do not need DMA backend
};
ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &tx_chan));
```

Install RMT RX Channel To install an RMT RX channel, there is a configuration structure that needs to be given in advance *rmt_rx_channel_config_t*. The following list describes each member of the configuration structure.

- *rmt_rx_channel_config_t::gpio_num* sets the GPIO number used by the receiver.
- *rmt_rx_channel_config_t::clk_src* selects the source clock for the RMT channel. The available clocks are listed in *rmt_clock_source_t*. Note that, the selected clock is also used by other channels, which means the user should ensure this configuration is the same when allocating other channels, regardless of TX or RX. For the effect on the power consumption of different clock sources, please refer to the *Power Management* section.
- *rmt_rx_channel_config_t::resolution_hz* sets the resolution of the internal tick counter. The timing parameter of the RMT signal is calculated based on this **tick**.
- *rmt_rx_channel_config_t::mem_block_symbols* has a slightly different meaning based on whether the DMA backend is enabled.
  - If the DMA is enabled via *rmt_rx_channel_config_t::with_dma*, this field controls the maximum size of the DMA buffer.
  - If DMA is not used, this field controls the size of the dedicated memory block owned by the channel, which should be at least 64.
- *rmt_rx_channel_config_t::invert_in* is used to invert the input signals before it is passed to the RMT receiver. The inversion is done by the GPIO matrix instead of by the RMT peripheral.
- *rmt_rx_channel_config_t::with_dma* enables the DMA backend for the channel. Using the DMA allows a significant amount of the channel's workload to be offloaded from the CPU. However, the DMA backend is not available on all ESP chips, please refer to [TRM] before you enable this option. Or you might encounter a *ESP_ERR_NOT_SUPPORTED* error.
- *rmt_rx_channel_config_t::io_loop_back* enables both input and output capabilities on the channel's assigned GPIO. Thus, by binding a TX and RX channel to the same GPIO, loopback can be achieved.
- *rmt_rx_channel_config_t::intr_priority* Set the priority of the interrupt. If set to `0` , then the

driver will use a interrupt with low or medium priority (priority level may be one of 1,2 or 3), otherwise use the priority indicated by *rmt_rx_channel_config_t::intr_priority*. Please use the number form (1,2,3) , not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until *rmt_del_channel()* is called.

Once the *rmt_rx_channel_config_t* structure is populated with mandatory parameters, users can call *rmt_new_rx_channel()* to allocate and initialize an RX channel. This function returns an RMT channel handle if it runs correctly. Specifically, when there are no more free channels in the RMT resource pool, this function returns *ESP_ERR_NOT_FOUND* error. If some feature (e.g., DMA backend) is not supported by the hardware, it returns *ESP_ERR_NOT_SUPPORTED* error.

```
rmt_channel_handle_t rx_chan = NULL;
rmt_rx_channel_config_t rx_chan_config = {
    .clk_src = RMT_CLK_SRC_DEFAULT,   // select source clock
    .resolution_hz = 1 * 1000 * 1000, // 1 MHz tick resolution, i.e., 1 tick = 1 µs
    .mem_block_symbols = 64,          // memory block size, 64 * 4 = 256 Bytes
    .gpio_num = 2,                    // GPIO number
    .flags.invert_in = false,         // do not invert input signal
    .flags.with_dma = false,          // do not need DMA backend
};
ESP_ERROR_CHECK(rmt_new_rx_channel(&rx_chan_config, &rx_chan));
```

Note: Due to a software limitation in the GPIO driver, when both TX and RX channels are bound to the same GPIO, ensure the RX Channel is initialized before the TX Channel. If the TX Channel was set up first, then during the RX Channel setup, the previous RMT TX Channel signal will be overridden by the GPIO control signal.

Uninstall RMT Channel If a previously installed RMT channel is no longer needed, it is recommended to recycle the resources by calling *rmt_del_channel()*, which in return allows the underlying software and hardware resources to be reused for other purposes.

Carrier Modulation and Demodulation The RMT transmitter can generate a carrier wave and modulate it onto the message signal. Compared to the message signal, the carrier signal ' s frequency is significantly higher. In addition, the user can only set the frequency and duty cycle for the carrier signal. The RMT receiver can demodulate the carrier signal from the incoming signal. Note that, carrier modulation and demodulation are not supported on all ESP chips, please refer to [TRM] before configuring the carrier, or you might encounter a *ESP_ERR_NOT_SUPPORTED* error.

Carrier-related configurations lie in *rmt_carrier_config_t*:

- *rmt_carrier_config_t::frequency_hz* sets the carrier frequency, in Hz.
- *rmt_carrier_config_t::duty_cycle* sets the carrier duty cycle.
- *rmt_carrier_config_t::polarity_active_low* sets the carrier polarity, i.e., on which level the carrier is applied.
- *rmt_carrier_config_t::always_on* sets whether to output the carrier even when the data transmission has finished. This configuration is only valid for the TX channel.

Note: For the RX channel, we should not set the carrier frequency exactly to the theoretical value. It is recommended to leave a tolerance for the carrier frequency. For example, in the snippet below, we set the frequency to 25 KHz, instead of the 38 KHz configured on the TX side. The reason is that reflection and refraction occur when a signal travels through the air, leading to distortion on the receiver side.

```
rmt_carrier_config_t tx_carrier_cfg = {
    .duty_cycle = 0.33,                 // duty cycle 33%
    .frequency_hz = 38000,              // 38 KHz
    .flags.polarity_active_low = false, // carrier should be modulated to high␣
↪level
};
```

(continues on next page)

(continued from previous page)

```
// modulate carrier to TX channel
ESP_ERROR_CHECK(rmt_apply_carrier(tx_chan, &tx_carrier_cfg));

rmt_carrier_config_t rx_carrier_cfg = {
    .duty_cycle = 0.33,                 // duty cycle 33%
    .frequency_hz = 25000,              // 25 KHz carrier, should be smaller than␣
↪the transmitter's carrier frequency
    .flags.polarity_active_low = false, // the carrier is modulated to high level
};
// demodulate carrier from RX channel
ESP_ERROR_CHECK(rmt_apply_carrier(rx_chan, &rx_carrier_cfg));
```

Register Event Callbacks When an event occurs on an RMT channel (e.g., transmission or receiving is completed), the CPU is notified of this event via an interrupt. If you have some function that needs to be called when a particular events occur, you can register a callback for that event to the RMT driver' s ISR (Interrupt Service Routine) by calling *rmt_tx_register_event_callbacks()* and *rmt_rx_register_event_callbacks()* for TX and RX channel respectively. Since the registered callback functions are called in the interrupt context, the user should ensure the callback function does not block, e.g., by making sure that only FreeRTOS APIs with the `FromISR` suffix are called from within the function. The callback function has a boolean return value used to indicate whether a higher priority task has been unblocked by the callback.

The TX channel-supported event callbacks are listed in the *rmt_tx_event_callbacks_t*:

- *rmt_tx_event_callbacks_t::on_trans_done* sets a callback function for the "trans-done"event. The function prototype is declared in *rmt_tx_done_callback_t*.

The RX channel-supported event callbacks are listed in the *rmt_rx_event_callbacks_t*:

- *rmt_rx_event_callbacks_t::on_recv_done* sets a callback function for "receive-done" event. The function prototype is declared in *rmt_rx_done_callback_t*.

Users can save their own context in *rmt_tx_register_event_callbacks()* and *rmt_rx_register_event_callbacks()* as well, via the parameter `user_data`. The user data is directly passed to each callback function.

In the callback function, users can fetch the event-specific data that is filled by the driver in the `edata`. Note that the `edata` pointer is only valid during the callback.

The TX-done event data is defined in *rmt_tx_done_event_data_t*:

- *rmt_tx_done_event_data_t::num_symbols* indicates the number of transmitted RMT symbols. This also reflects the size of the encoding artifacts. Please note, this value accounts for the `EOF` symbol as well, which is appended by the driver to mark the end of one transaction.

The RX-complete event data is defined in *rmt_rx_done_event_data_t*:

- *rmt_rx_done_event_data_t::received_symbols* points to the received RMT symbols. These symbols are saved in the `buffer` parameter of the *rmt_receive()* function. Users should not free this receive buffer before the callback returns.
- *rmt_rx_done_event_data_t::num_symbols* indicates the number of received RMT symbols. This value is not larger than the `buffer_size` parameter of *rmt_receive()* function. If the `buffer_size` is not sufficient to accommodate all the received RMT symbols, the driver only keeps the maximum number of symbols that the buffer can hold, and excess symbols are discarded or ignored.

Enable and Disable Channel *rmt_enable()* must be called in advance before transmitting or receiving RMT symbols. For TX channels, enabling a channel enables a specific interrupt and prepares the hardware to dispatch transactions. For RX channels, enabling a channel enables an interrupt, but the receiver is not started during this time, as the characteristics of the incoming signal have yet to be specified. The receiver is started in *rmt_receive()*.

rmt_disable() does the opposite by disabling the interrupt and clearing any pending interrupts. The transmitter and receiver are disabled as well.

```
ESP_ERROR_CHECK(rmt_enable(tx_chan));
ESP_ERROR_CHECK(rmt_enable(rx_chan));
```

Initiate TX Transaction RMT is a special communication peripheral, as it is unable to transmit raw byte streams like SPI and I2C. RMT can only send data in its own format *rmt_symbol_word_t*. However, the hardware does not help to convert the user data into RMT symbols, this can only be done in software by the so-called **RMT Encoder**. The encoder is responsible for encoding user data into RMT symbols and then writing to the RMT memory block or the DMA buffer. For how to create an RMT encoder, please refer to *RMT Encoder*.

Once you created an encoder, you can initiate a TX transaction by calling *rmt_transmit()*. This function takes several positional parameters like channel handle, encoder handle, and payload buffer. Besides, you also need to provide a transmission-specific configuration in *rmt_transmit_config_t*:

- *rmt_transmit_config_t::loop_count* sets the number of transmission loops. After the transmitter has finished one round of transmission, it can restart the same transmission again if this value is not set to zero. As the loop is controlled by hardware, the RMT channel can be used to generate many periodic sequences with minimal CPU intervention.
  - Setting *rmt_transmit_config_t::loop_count* to *-1* means an infinite loop transmission. In this case, the channel does not stop until *rmt_disable()* is called. The "trans-done" event is not generated as well.
  - Setting *rmt_transmit_config_t::loop_count* to a positive number means finite number of iterations. In this case, the "trans-done" event is when the specified number of iterations have completed.

  Note: The **loop transmit** feature is not supported on all ESP chips, please refer to [TRM] before you configure this option, or you might encounter *ESP_ERR_NOT_SUPPORTED* error.

- *rmt_transmit_config_t::eot_level* sets the output level when the transmitter finishes working or stops working by calling *rmt_disable()*.
- *rmt_transmit_config_t::queue_nonblocking* sets whether to wait for a free slot in the transaction queue when it is full. If this value is set to `true`, then the function will return with an error code *ESP_ERR_INVALID_STATE* when the queue is full. Otherwise, the function will block until a free slot is available in the queue.

Note: There is a limitation in the transmission size if the *rmt_transmit_config_t::loop_count* is set to non-zero, i.e., to enable the loop feature. The encoded RMT symbols should not exceed the capacity of the RMT hardware memory block size, or you might see an error message like `encoding artifacts can't exceed hw memory block for loop transmission`. If you have to start a large transaction by loop, you can try either of the following methods.

- Increase the *rmt_tx_channel_config_t::mem_block_symbols*. This approach does not work if the DMA backend is also enabled.
- Customize an encoder and construct an infinite loop in the encoding function. See also *RMT Encoder*.

Internally, *rmt_transmit()* constructs a transaction descriptor and sends it to a job queue, which is dispatched in the ISR. So it is possible that the transaction is not started yet when *rmt_transmit()* returns. To ensure all pending transactions to complete, the user can use *rmt_tx_wait_all_done()*.

Multiple Channels Simultaneous Transmission In some real-time control applications (e.g., to make two robotic arms move simultaneously), you do not want any time drift between different channels. The RMT driver can help to manage this by creating a so-called **Sync Manager**. The sync manager is represented by *rmt_sync_manager_handle_t* in the driver. The procedure of RMT sync transmission is shown as follows:

Install RMT Sync Manager To create a sync manager, the user needs to tell which channels are going to be managed in the *rmt_sync_manager_config_t*:

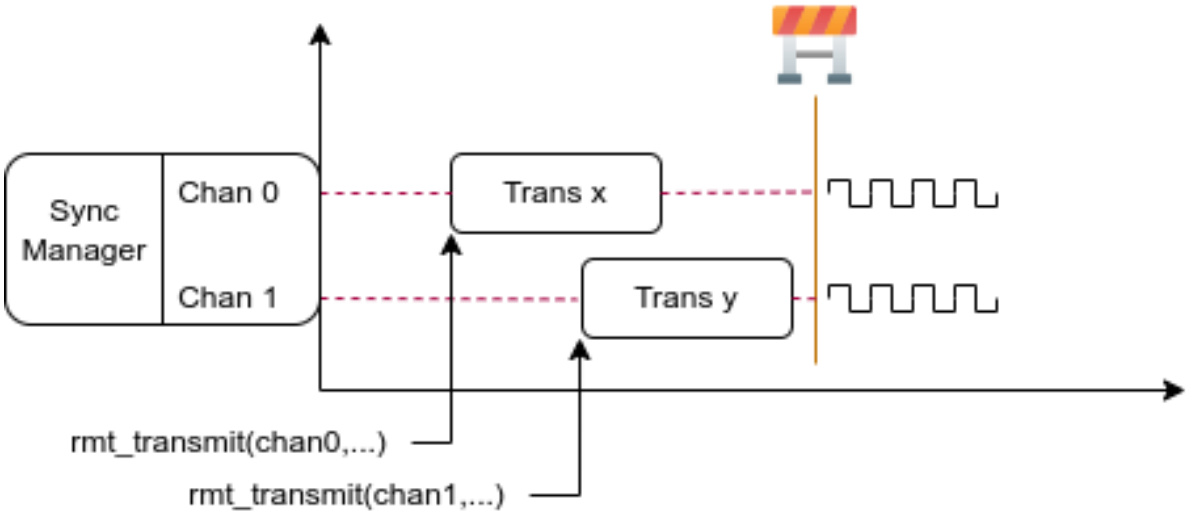


Fig. 18: RMT TX Sync

- *rmt_sync_manager_config_t::tx_channel_array* points to the array of TX channels to be managed.
- *rmt_sync_manager_config_t::array_size* sets the number of channels to be managed.

rmt_new_sync_manager() can return a manager handle on success. This function could also fail due to various errors such as invalid arguments, etc. Especially, when the sync manager has been installed before, and there are no hardware resources to create another manager, this function reports *ESP_ERR_NOT_FOUND* error. In addition, if the sync manager is not supported by the hardware, it reports a *ESP_ERR_NOT_SUPPORTED* error. Please refer to [TRM] before using the sync manager feature.

Start Transmission Simultaneously For any managed TX channel, it does not start the machine until *rmt_transmit()* has been called on all channels in *rmt_sync_manager_config_t::tx_channel_array*. Before that, the channel is just put in a waiting state. TX channels will usually complete their transactions at different times due to differing transactions, thus resulting in a loss of sync. So before restarting a simultaneous transmission, the user needs to call *rmt_sync_reset()* to synchronize all channels again.

Calling *rmt_del_sync_manager()* can recycle the sync manager and enable the channels to initiate transactions independently afterward.

```
rmt_channel_handle_t tx_channels[2] = {NULL}; // declare two channels
int tx_gpio_number[2] = {0, 2};
// install channels one by one
for (int i = 0; i < 2; i++) {
    rmt_tx_channel_config_t tx_chan_config = {
        .clk_src = RMT_CLK_SRC_DEFAULT,        // select source clock
        .gpio_num = tx_gpio_number[i],     // GPIO number
        .mem_block_symbols = 64,           // memory block size, 64 * 4 = 256 Bytes
        .resolution_hz = 1 * 1000 * 1000, // 1 MHz resolution
        .trans_queue_depth = 1,            // set the number of transactions that␣
↪can pend in the background
    };
    ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &tx_channels[i]));
}
// install sync manager
rmt_sync_manager_handle_t synchro = NULL;
rmt_sync_manager_config_t synchro_config = {
    .tx_channel_array = tx_channels,
    .array_size = sizeof(tx_channels) / sizeof(tx_channels[0]),
```

(continues on next page)

(continued from previous page)

```
};
ESP_ERROR_CHECK(rmt_new_sync_manager(&synchro_config, &synchro));

ESP_ERROR_CHECK(rmt_transmit(tx_channels[0], led_strip_encoders[0], led_data, led_
↪num * 3, &transmit_config));
// tx_channels[0] does not start transmission until call of `rmt_transmit()` for␣
↪tx_channels[1] returns
ESP_ERROR_CHECK(rmt_transmit(tx_channels[1], led_strip_encoders[1], led_data, led_
↪num * 3, &transmit_config));
```

Initiate RX Transaction As also discussed in the *Enable and Disable Channel*, calling `rmt_enable()` does not prepare an RX to receive RMT symbols. The user needs to specify the basic characteristics of the incoming signals in `rmt_receive_config_t`:

- `rmt_receive_config_t::signal_range_min_ns` specifies the minimal valid pulse duration in either high or low logic levels. A pulse width that is smaller than this value is treated as a glitch, and ignored by the hardware.
- `rmt_receive_config_t::signal_range_max_ns` specifies the maximum valid pulse duration in either high or low logic levels. A pulse width that is bigger than this value is treated as **Stop Signal**, and the receiver generates receive-complete event immediately.

The RMT receiver starts the RX machine after the user calls `rmt_receive()` with the provided configuration above. Note that, this configuration is transaction specific, which means, to start a new round of reception, the user needs to set the `rmt_receive_config_t` again. The receiver saves the incoming signals into its internal memory block or DMA buffer, in the format of `rmt_symbol_word_t`.

Due to the limited size of the memory block, the RMT receiver can only save short frames whose length is not longer than the memory block capacity. Long frames are truncated by the hardware, and the driver reports an error message: `hw buffer too small, received symbols truncated`.

The copy destination should be provided in the `buffer` parameter of `rmt_receive()` function. If this buffer overlfows due to an insufficient buffer size, the receiver can continue to work, but overflowed symbols are dropped and the following error message is reported: `user buffer too small, received symbols truncated`. Please take care of the lifecycle of the `buffer` parameter, ensuring that the buffer is not recycled before the receiver is finished or stopped.

The receiver is stopped by the driver when it finishes working, i.e., receive a signal whose duration is bigger than `rmt_receive_config_t::signal_range_max_ns`. The user needs to call `rmt_receive()` again to restart the receiver, if necessary. The user can get the received data in the `rmt_rx_event_callbacks_t::on_recv_done` callback. See also *Register Event Callbacks* for more information.

```
static bool example_rmt_rx_done_callback(rmt_channel_handle_t channel, const rmt_
↪rx_done_event_data_t *edata, void *user_data)
{
    BaseType_t high_task_wakeup = pdFALSE;
    QueueHandle_t receive_queue = (QueueHandle_t)user_data;
    // send the received RMT symbols to the parser task
    xQueueSendFromISR(receive_queue, edata, &high_task_wakeup);
    // return whether any task is woken up
    return high_task_wakeup == pdTRUE;
}

QueueHandle_t receive_queue = xQueueCreate(1, sizeof(rmt_rx_done_event_data_t));
rmt_rx_event_callbacks_t cbs = {
    .on_recv_done = example_rmt_rx_done_callback,
};
ESP_ERROR_CHECK(rmt_rx_register_event_callbacks(rx_channel, &cbs, receive_queue));

// the following timing requirement is based on NEC protocol
```

(continues on next page)

(continued from previous page)

```
rmt_receive_config_t receive_config = {
    .signal_range_min_ns = 1250,     // the shortest duration for NEC signal is␣
↪560 µs, 1250 ns < 560 µs, valid signal is not treated as noise
    .signal_range_max_ns = 12000000, // the longest duration for NEC signal is␣
↪9000 µs, 12000000 ns > 9000 µs, the receive does not stop early
};

rmt_symbol_word_t raw_symbols[64]; // 64 symbols should be sufficient for a␣
↪standard NEC frame
// ready to receive
ESP_ERROR_CHECK(rmt_receive(rx_channel, raw_symbols, sizeof(raw_symbols), &receive_
↪config));
// wait for the RX-done signal
rmt_rx_done_event_data_t rx_data;
xQueueReceive(receive_queue, &rx_data, portMAX_DELAY);
// parse the received symbols
example_parse_nec_frame(rx_data.received_symbols, rx_data.num_symbols);
```

RMT Encoder An RMT encoder is part of the RMT TX transaction, whose responsibility is to generate and write the correct RMT symbols into hardware memory or DMA buffer at a specific time. There are some special restrictions for an encoding function:

- During a single transaction, the encoding function may be called multiple times. This is necessary because the target RMT memory block cannot hold all the artifacts at once. To overcome this limitation, the driver utilizes a **ping-pong** approach, where the encoding session is divided into multiple parts. This means that the encoder needs to **keep track of its state** to continue encoding from where it left off in the previous part.
- The encoding function is running in the ISR context. To speed up the encoding session, it is highly recommended to put the encoding function into IRAM. This can also avoid the cache miss during encoding.

To help get started with the RMT driver faster, some commonly used encoders are provided out-of-the-box. They can either work alone or be chained together into a new encoder. See also Composite Pattern for the principle behind it. The driver has defined the encoder interface in *rmt_encoder_t*, it contains the following functions:

- *rmt_encoder_t::encode* is the fundamental function of an encoder. This is where the encoding session happens.
  - The function might be called multiple times within a single transaction. The encode function should return the state of the current encoding session.
  - The supported states are listed in the *rmt_encode_state_t*. If the result contains *RMT_ENCODING_COMPLETE*, it means the current encoder has finished work.
  - If the result contains *RMT_ENCODING_MEM_FULL*, the program needs to yield from the current session, as there is no space to save more encoding artifacts.
- *rmt_encoder_t::reset* should reset the encoder state back to the initial state (the RMT encoder is stateful).
  - If the RMT transmitter is manually stopped without resetting its corresponding encoder, subsequent encoding session can be erroneous.
  - This function is also called implicitly in *rmt_disable()*.
- *rmt_encoder_t::del* should free the resources allocated by the encoder.

Copy Encoder A copy encoder is created by calling *rmt_new_copy_encoder()*. A copy encoder's main functionality is to copy the RMT symbols from user space into the driver layer. It is usually used to encode `const` data, i.e., data does not change at runtime after initialization such as the leading code in the IR protocol.

A configuration structure *rmt_copy_encoder_config_t* should be provided in advance before calling *rmt_new_copy_encoder()*. Currently, this configuration is reserved for future expansion, and has no specific use or setting items for now.

Bytes Encoder A bytes encoder is created by calling *rmt_new_bytes_encoder()*. The bytes encoder's main functionality is to convert the user space byte stream into RMT symbols dynamically. It is usually used to

encode dynamic data, e.g., the address and command fields in the IR protocol.

A configuration structure *rmt_bytes_encoder_config_t* should be provided in advance before calling *rmt_new_bytes_encoder()*:

- *rmt_bytes_encoder_config_t::bit0* and *rmt_bytes_encoder_config_t::bit1* are necessary to specify the encoder how to represent bit zero and bit one in the format of *rmt_symbol_word_t*.
- *rmt_bytes_encoder_config_t::msb_first* sets the bit endianess of each byte. If it is set to true, the encoder encodes the **Most Significant Bit** first. Otherwise, it encodes the **Least Significant Bit** first.

Besides the primitive encoders provided by the driver, the user can implement his own encoder by chaining the existing encoders together. A common encoder chain is shown as follows:

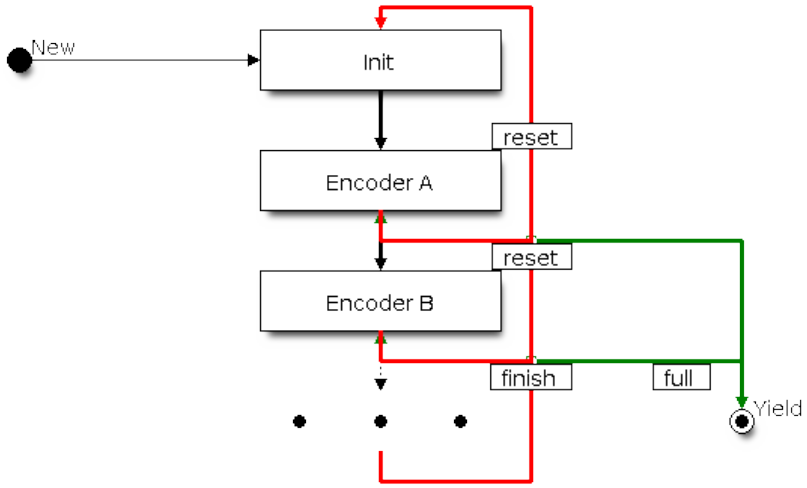


Fig. 19: RMT Encoder Chain

Customize RMT Encoder for NEC Protocol This section demonstrates how to write an NEC encoder. The NEC IR protocol uses pulse distance encoding of the message bits. Each pulse burst is `562.5 µs` in length, logical bits are transmitted as follows. It is worth mentioning that the least significant bit of each byte is sent first.

- Logical `0`: a `562.5 µs` pulse burst followed by a `562.5 µs` space, with a total transmit time of `1.125 ms`
- Logical `1`: a `562.5 µs` pulse burst followed by a `1.6875 ms` space, with a total transmit time of `2.25 ms`

When a key is pressed on the remote controller, the transmitted message includes the following elements in the specified order:

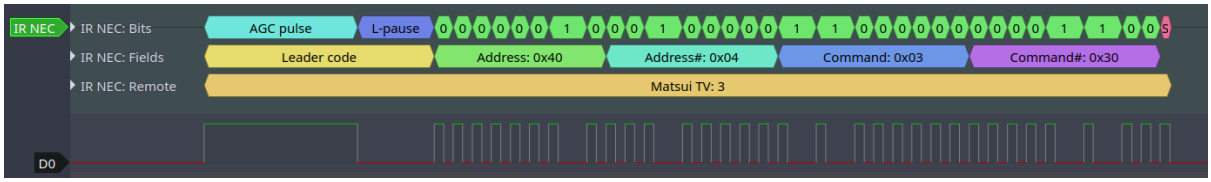


Fig. 20: IR NEC Frame

- `9 ms` leading pulse burst, also called the "AGC pulse"
- `4.5 ms` space
- 8-bit address for the receiving device
- 8-bit logical inverse of the address
- 8-bit command

- 8-bit logical inverse of the command
- a final 562.5 µs pulse burst to signify the end of message transmission

Then you can construct the NEC *rmt_encoder_t::encode* function in the same order, for example:

```
// IR NEC scan code representation
typedef struct {
    uint16_t address;
    uint16_t command;
} ir_nec_scan_code_t;

// construct an encoder by combining primitive encoders
typedef struct {
    rmt_encoder_t base;           // the base "class" declares the standard␣
↪encoder interface
    rmt_encoder_t *copy_encoder;  // use the copy_encoder to encode the leading␣
↪and ending pulse
    rmt_encoder_t *bytes_encoder; // use the bytes_encoder to encode the address␣
↪and command data
    rmt_symbol_word_t nec_leading_symbol; // NEC leading code with RMT␣
↪representation
    rmt_symbol_word_t nec_ending_symbol;  // NEC ending code with RMT␣
↪representation
    int state; // record the current encoding state, i.e., we are in which␣
↪encoding phase
} rmt_ir_nec_encoder_t;

static size_t rmt_encode_ir_nec(rmt_encoder_t *encoder, rmt_channel_handle_t␣
↪channel, const void *primary_data, size_t data_size, rmt_encode_state_t *ret_
↪state)
{
    rmt_ir_nec_encoder_t *nec_encoder = __containerof(encoder, rmt_ir_nec_encoder_
↪t, base);
    rmt_encode_state_t session_state = RMT_ENCODING_RESET;
    rmt_encode_state_t state = RMT_ENCODING_RESET;
    size_t encoded_symbols = 0;
    ir_nec_scan_code_t *scan_code = (ir_nec_scan_code_t *)primary_data;
    rmt_encoder_handle_t copy_encoder = nec_encoder->copy_encoder;
    rmt_encoder_handle_t bytes_encoder = nec_encoder->bytes_encoder;
    switch (nec_encoder->state) {
    case 0: // send leading code
        encoded_symbols += copy_encoder->encode(copy_encoder, channel, &nec_
↪encoder->nec_leading_symbol,
                                                sizeof(rmt_symbol_word_t), &
↪session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = 1; // we can only switch to the next state when␣
↪the current encoder finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding␣
↪artifacts
        }
    // fall-through
    case 1: // send address
        encoded_symbols += bytes_encoder->encode(bytes_encoder, channel, &scan_
↪code->address, sizeof(uint16_t), &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = 2; // we can only switch to the next state when␣
↪the current encoder finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
```

(continues on next page)

(continued from previous page)

```
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding␣
↪artifacts
        }
    // fall-through
    case 2: // send command
        encoded_symbols += bytes_encoder->encode(bytes_encoder, channel, &scan_
↪code->command, sizeof(uint16_t), &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = 3; // we can only switch to the next state when␣
↪the current encoder finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding␣
↪artifacts
        }
    // fall-through
    case 3: // send ending code
        encoded_symbols += copy_encoder->encode(copy_encoder, channel, &nec_
↪encoder->nec_ending_symbol,
                                                sizeof(rmt_symbol_word_t), &
↪session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = RMT_ENCODING_RESET; // back to the initial␣
↪encoding session
            state |= RMT_ENCODING_COMPLETE; // telling the caller the NEC encoding␣
↪has finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding␣
↪artifacts
        }
    }
out:
    *ret_state = state;
    return encoded_symbols;
}
```

A full sample code can be found in peripherals/rmt/ir_nec_transceiver. In the above snippet, we use a `switch-case` and several `goto` statements to implement a Finite-state machine . With this pattern, users can construct much more complex IR protocols.

Power Management When power management is enabled, i.e., *CONFIG_PM_ENABLE* is on, the system adjusts the APB frequency before going into Light-sleep, thus potentially changing the resolution of the RMT internal counter.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type *ESP_PM_APB_FREQ_MAX*. Whenever the user creates an RMT channel that has selected *RMT_CLK_SRC_APB* as the clock source, the driver guarantees that the power management lock is acquired after the channel enabled by *rmt_enable()*. Likewise, the driver releases the lock after *rmt_disable()* is called for the same channel. This also reveals that the *rmt_enable()* and *rmt_disable()* should appear in pairs.

If the channel clock source is selected to others like `RMT_CLK_SRC_XTAL`, then the driver does not install a power management lock for it, which is more suitable for a low-power application as long as the source clock can still provide sufficient resolution.

IRAM Safe By default, the RMT interrupt is deferred when the Cache is disabled for reasons like writing or erasing the main Flash. Thus the transaction-done interrupt does not get handled in time, which is not acceptable in a real-

time application. What is worse, when the RMT transaction relies on **ping-pong** interrupt to successively encode or copy RMT symbols, a delayed interrupt can lead to an unpredictable result.

There is a Kconfig option *CONFIG_RMT_ISR_IRAM_SAFE* that has the following features:

1. Enable the interrupt being serviced even when the cache is disabled
2. Place all functions used by the ISR into IRAM[2]
3. Place the driver object into DRAM in case it is mapped to PSRAM by accident

This Kconfig option allows the interrupt handler to run while the cache is disabled but comes at the cost of increased IRAM consumption.

Another Kconfig option *CONFIG_RMT_RECV_FUNC_IN_IRAM* can place `rmt_receive()` into the IRAM as well. So that the receive function can be used even when the flash cache is disabled.

Thread Safety The factory function `rmt_new_tx_channel()`, `rmt_new_rx_channel()` and `rmt_new_sync_manager()` are guaranteed to be thread-safe by the driver, which means, user can call them from different RTOS tasks without protection by extra locks. Other functions that take the `rmt_channel_handle_t` and `rmt_sync_manager_handle_t` as the first positional parameter, are not thread-safe. which means the user should avoid calling them from multiple tasks.

The following functions are allowed to use under ISR context as well.

- `rmt_receive()`

Kconfig Options

- *CONFIG_RMT_ISR_IRAM_SAFE* controls whether the default ISR handler can work when cache is disabled, see also *IRAM Safe* for more information.
- *CONFIG_RMT_ENABLE_DEBUG_LOG* is used to enable the debug log at the cost of increased firmware binary size.
- *CONFIG_RMT_RECV_FUNC_IN_IRAM* controls where to place the RMT receive function (IRAM or Flash), see *IRAM Safe* for more information.

Application Examples

- RMT-based RGB LED strip customized encoder: peripherals/rmt/led_strip
- RMT IR NEC protocol encoding and decoding: peripherals/rmt/ir_nec_transceiver
- RMT transactions in queue: peripherals/rmt/musical_buzzer
- RMT-based stepper motor with S-curve algorithm: : peripherals/rmt/stepper_motor
- RMT infinite loop for driving DShot ESC: peripherals/rmt/dshot_esc
- RMT simulate 1-wire protocol (take DS18B20 as example): peripherals/rmt/onewire

FAQ

- Why the RMT encoder results in more data than expected?

The RMT encoding takes place in the ISR context. If your RMT encoding session takes a long time (e.g., by logging debug information) or the encoding session is deferred somehow because of interrupt latency, then it is possible the transmitting becomes **faster** than the encoding. As a result, the encoder can not prepare the next data in time, leading to the transmitter sending the previous data again. There is no way to ask the transmitter to stop and wait. You can mitigate the issue by combining the following ways:

- Increase the `rmt_tx_channel_config_t::mem_block_symbols`, in steps of 64.
- Place the encoding function in the IRAM.
- Enables the `rmt_tx_channel_config_t::with_dma` if it is available for your chip.

[2] The callback function, e.g., `rmt_tx_event_callbacks_t::on_trans_done`, and the functions invoked by itself should also reside in IRAM, users need to take care of this by themselves.

API Reference

Header File

- components/driver/include/driver/rmt_tx.h

Functions

esp_err_t **rmt_new_tx_channel**(const *rmt_tx_channel_config_t* *config, *rmt_channel_handle_t* *ret_chan)

Create a RMT TX channel.

Parameters

- **config** –**[in]** TX channel configurations
- **ret_chan** –**[out]** Returned generic RMT channel handle

Returns

- ESP_OK: Create RMT TX channel successfully
- ESP_ERR_INVALID_ARG: Create RMT TX channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create RMT TX channel failed because out of memory
- ESP_ERR_NOT_FOUND: Create RMT TX channel failed because all RMT channels are used up and no more free one
- ESP_ERR_NOT_SUPPORTED: Create RMT TX channel failed because some feature is not supported by hardware, e.g. DMA feature is not supported by hardware
- ESP_FAIL: Create RMT TX channel failed because of other error

esp_err_t **rmt_transmit**(*rmt_channel_handle_t* tx_channel, *rmt_encoder_handle_t* encoder, const void *payload, size_t payload_bytes, const *rmt_transmit_config_t* *config)

Transmit data by RMT TX channel.

Note: This function constructs a transaction descriptor then pushes to a queue. The transaction will not start immediately if there's another one under processing. Based on the setting of *rmt_transmit_config_t::queue_nonblocking*, if there're too many transactions pending in the queue, this function can block until it has free slot, otherwise just return quickly.

Note: The data to be transmitted will be encoded into RMT symbols by the specific `encoder`.

Parameters

- **tx_channel** –**[in]** RMT TX channel that created by `rmt_new_tx_channel()`
- **encoder** –**[in]** RMT encoder that created by various factory APIs like `rmt_new_bytes_encoder()`
- **payload** –**[in]** The raw data to be encoded into RMT symbols
- **payload_bytes** –**[in]** Size of the `payload` in bytes
- **config** –**[in]** Transmission specific configuration

Returns

- ESP_OK: Transmit data successfully
- ESP_ERR_INVALID_ARG: Transmit data failed because of invalid argument
- ESP_ERR_INVALID_STATE: Transmit data failed because channel is not enabled
- ESP_ERR_NOT_SUPPORTED: Transmit data failed because some feature is not supported by hardware, e.g. unsupported loop count
- ESP_FAIL: Transmit data failed because of other error

esp_err_t **rmt_tx_wait_all_done**(*rmt_channel_handle_t* tx_channel, int timeout_ms)

Wait for all pending TX transactions done.

Note: This function will block forever if the pending transaction can't be finished within a limited time (e.g. an infinite loop transaction). See also `rmt_disable()` for how to terminate a working channel.

Parameters

- **tx_channel** –**[in]** RMT TX channel that created by `rmt_new_tx_channel()`
- **timeout_ms** –**[in]** Wait timeout, in ms. Specially, -1 means to wait forever.

Returns

- ESP_OK: Flush transactions successfully
- ESP_ERR_INVALID_ARG: Flush transactions failed because of invalid argument
- ESP_ERR_TIMEOUT: Flush transactions failed because of timeout
- ESP_FAIL: Flush transactions failed because of other error

esp_err_t **rmt_tx_register_event_callbacks**(*rmt_channel_handle_t* tx_channel, const *rmt_tx_event_callbacks_t* *cbs, void *user_data)

Set event callbacks for RMT TX channel.

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Note: When CONFIG_RMT_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

Parameters

- **tx_channel** –**[in]** RMT generic channel that created by `rmt_new_tx_channel()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

esp_err_t **rmt_new_sync_manager**(const *rmt_sync_manager_config_t* *config, *rmt_sync_manager_handle_t* *ret_synchro)

Create a synchronization manager for multiple TX channels, so that the managed channel can start transmitting at the same time.

Note: All the channels to be managed should be enabled by `rmt_enable()` before put them into sync manager.

Parameters

- **config** –**[in]** Synchronization manager configuration
- **ret_synchro** –**[out]** Returned synchronization manager handle

Returns

- ESP_OK: Create sync manager successfully
- ESP_ERR_INVALID_ARG: Create sync manager failed because of invalid argument
- ESP_ERR_NOT_SUPPORTED: Create sync manager failed because it is not supported by hardware
- ESP_ERR_INVALID_STATE: Create sync manager failed because not all channels are enabled
- ESP_ERR_NO_MEM: Create sync manager failed because out of memory
- ESP_ERR_NOT_FOUND: Create sync manager failed because all sync controllers are used up and no more free one
- ESP_FAIL: Create sync manager failed because of other error

esp_err_t **rmt_del_sync_manager**(*rmt_sync_manager_handle_t* synchro)

Delete synchronization manager.

Parameters **synchro** –**[in]** Synchronization manager handle returned from `rmt_new_sync_manager()`

Returns

- ESP_OK: Delete the synchronization manager successfully
- ESP_ERR_INVALID_ARG: Delete the synchronization manager failed because of invalid argument
- ESP_FAIL: Delete the synchronization manager failed because of other error

esp_err_t **rmt_sync_reset**(*rmt_sync_manager_handle_t* synchro)

Reset synchronization manager.

Parameters **synchro** –**[in]** Synchronization manager handle returned from `rmt_new_sync_manager()`

Returns

- ESP_OK: Reset the synchronization manager successfully
- ESP_ERR_INVALID_ARG: Reset the synchronization manager failed because of invalid argument
- ESP_FAIL: Reset the synchronization manager failed because of other error

Structures

struct **rmt_tx_event_callbacks_t**

Group of RMT TX callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_RMT_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

rmt_tx_done_callback_t **on_trans_done**

Event callback, invoked when transmission is finished

struct **rmt_tx_channel_config_t**

RMT TX channel specific configuration.

Public Members

gpio_num_t **gpio_num**

GPIO number used by RMT TX channel. Set to -1 if unused

rmt_clock_source_t **clk_src**

Clock source of RMT TX channel, channels in the same group must use the same clock source

uint32_t **resolution_hz**

Channel clock resolution, in Hz

size_t **mem_block_symbols**

Size of memory block, in number of *rmt_symbol_word_t*, must be an even. In the DMA mode, this field controls the DMA buffer size, it can be set to a large value; In the normal mode, this field controls the number of RMT memory block that will be used by the channel.

size_t **trans_queue_depth**

Depth of internal transfer queue, increase this value can support more transfers pending in the background

int **intr_priority**

RMT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

uint32_t **invert_out**

Whether to invert the RMT channel signal before output to GPIO pad

uint32_t **with_dma**

If set, the driver will allocate an RMT channel with DMA capability

uint32_t **io_loop_back**

The signal output from the GPIO will be fed to the input path as well

uint32_t **io_od_mode**

Configure the GPIO as open-drain mode

struct *rmt_tx_channel_config_t*::[anonymous] **flags**

TX channel config flags

struct **rmt_transmit_config_t**

RMT transmit specific configuration.

Public Members

int **loop_count**

Specify the times of transmission in a loop, -1 means transmitting in an infinite loop

uint32_t **eot_level**

Set the output level for the "End Of Transmission"

uint32_t **queue_nonblocking**

If set, when the transaction queue is full, driver will not block the thread but return directly

struct *rmt_transmit_config_t*::[anonymous] **flags**

Transmit specific config flags

struct **rmt_sync_manager_config_t**

Synchronous manager configuration.

Public Members

const *rmt_channel_handle_t* ***tx_channel_array**

Array of TX channels that are about to be managed by a synchronous controller

size_t **array_size**

Size of the `tx_channel_array`

Header File

- components/driver/include/driver/rmt_rx.h

Functions

esp_err_t **rmt_new_rx_channel** (const *rmt_rx_channel_config_t* *config, *rmt_channel_handle_t* *ret_chan)

Create a RMT RX channel.

Parameters

- **config** –**[in]** RX channel configurations
- **ret_chan** –**[out]** Returned generic RMT channel handle

Returns

- ESP_OK: Create RMT RX channel successfully
- ESP_ERR_INVALID_ARG: Create RMT RX channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create RMT RX channel failed because out of memory
- ESP_ERR_NOT_FOUND: Create RMT RX channel failed because all RMT channels are used up and no more free one
- ESP_ERR_NOT_SUPPORTED: Create RMT RX channel failed because some feature is not supported by hardware, e.g. DMA feature is not supported by hardware
- ESP_FAIL: Create RMT RX channel failed because of other error

esp_err_t **rmt_receive** (*rmt_channel_handle_t* rx_channel, void *buffer, size_t buffer_size, const *rmt_receive_config_t* *config)

Initiate a receive job for RMT RX channel.

Note: This function is non-blocking, it initiates a new receive job and then returns. User should check the received data from the `on_recv_done` callback that registered by `rmt_rx_register_event_callbacks()`.

Note: This function can also be called in ISR context.

Note: If you want this function to work even when the flash cache is disabled, please enable the `CONFIG_RMT_RECV_FUNC_IN_IRAM` option.

Parameters

- **rx_channel** –**[in]** RMT RX channel that created by `rmt_new_rx_channel()`
- **buffer** –**[in]** The buffer to store the received RMT symbols
- **buffer_size** –**[in]** size of the `buffer`, in bytes
- **config** –**[in]** Receive specific configurations

Returns

- ESP_OK: Initiate receive job successfully
- ESP_ERR_INVALID_ARG: Initiate receive job failed because of invalid argument
- ESP_ERR_INVALID_STATE: Initiate receive job failed because channel is not enabled
- ESP_FAIL: Initiate receive job failed because of other error

esp_err_t **rmt_rx_register_event_callbacks**(*rmt_channel_handle_t* rx_channel, const *rmt_rx_event_callbacks_t* *cbs, void *user_data)

Set callbacks for RMT RX channel.

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Note: When CONFIG_RMT_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

Parameters

- **rx_channel** –**[in]** RMT generic channel that created by `rmt_new_rx_channel()`
- **cbs** –**[in]** Group of callback functions
- **user_data** –**[in]** User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

Structures

struct **rmt_rx_event_callbacks_t**

Group of RMT RX callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_RMT_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

rmt_rx_done_callback_t **on_recv_done**

Event callback, invoked when one RMT channel receiving transaction completes

struct **rmt_rx_channel_config_t**

RMT RX channel specific configuration.

Public Members

gpio_num_t **gpio_num**

GPIO number used by RMT RX channel. Set to -1 if unused

rmt_clock_source_t **clk_src**

Clock source of RMT RX channel, channels in the same group must use the same clock source

uint32_t **resolution_hz**

Channel clock resolution, in Hz

size_t **mem_block_symbols**

Size of memory block, in number of *rmt_symbol_word_t*, must be an even. In the DMA mode, this field controls the DMA buffer size, it can be set to a large value (e.g. 1024); In the normal mode, this field controls the number of RMT memory block that will be used by the channel.

uint32_t **invert_in**

Whether to invert the incoming RMT channel signal

uint32_t **with_dma**

If set, the driver will allocate an RMT channel with DMA capability

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

struct *rmt_rx_channel_config_t*::[anonymous] **flags**

RX channel config flags

int **intr_priority**

RMT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

struct **rmt_receive_config_t**

RMT receive specific configuration.

Public Members

uint32_t **signal_range_min_ns**

A pulse whose width is smaller than this threshold will be treated as glitch and ignored

uint32_t **signal_range_max_ns**

RMT will stop receiving if one symbol level has kept more than `signal_range_max_ns`

Header File

- components/driver/include/driver/rmt_common.h

Functions

esp_err_t **rmt_del_channel**(*rmt_channel_handle_t* channel)

Delete an RMT channel.

Parameters **channel** –**[in]** RMT generic channel that created by `rmt_new_tx_channel()` or `rmt_new_rx_channel()`

Returns

- ESP_OK: Delete RMT channel successfully
- ESP_ERR_INVALID_ARG: Delete RMT channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete RMT channel failed because it is still in working
- ESP_FAIL: Delete RMT channel failed because of other error

esp_err_t **rmt_apply_carrier**(*rmt_channel_handle_t* channel, const *rmt_carrier_config_t* *config)

Apply modulation feature for TX channel or demodulation feature for RX channel.

Parameters

- **channel** –**[in]** RMT generic channel that created by `rmt_new_tx_channel()` or `rmt_new_rx_channel()`
- **config** –**[in]** Carrier configuration. Specially, a NULL config means to disable the carrier modulation or demodulation feature

Returns

- ESP_OK: Apply carrier configuration successfully
- ESP_ERR_INVALID_ARG: Apply carrier configuration failed because of invalid argument
- ESP_FAIL: Apply carrier configuration failed because of other error

esp_err_t **rmt_enable**(*rmt_channel_handle_t* channel)

Enable the RMT channel.

Note: This function will acquire a PM lock that might be installed during channel allocation

---

Parameters **channel** –**[in]** RMT generic channel that created by `rmt_new_tx_channel()` or `rmt_new_rx_channel()`

Returns

- ESP_OK: Enable RMT channel successfully
- ESP_ERR_INVALID_ARG: Enable RMT channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable RMT channel failed because it' s enabled already
- ESP_FAIL: Enable RMT channel failed because of other error

esp_err_t **rmt_disable**(*rmt_channel_handle_t* channel)

Disable the RMT channel.

Note: This function will release a PM lock that might be installed during channel allocation

---

Parameters **channel** –**[in]** RMT generic channel that created by `rmt_new_tx_channel()` or `rmt_new_rx_channel()`

Returns

- ESP_OK: Disable RMT channel successfully
- ESP_ERR_INVALID_ARG: Disable RMT channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable RMT channel failed because it' s not enabled yet
- ESP_FAIL: Disable RMT channel failed because of other error

Structures

struct **rmt_carrier_config_t**

RMT carrier wave configuration (for either modulation or demodulation)

Public Members

uint32_t **frequency_hz**

Carrier wave frequency, in Hz, 0 means disabling the carrier

float **duty_cycle**

Carrier wave duty cycle (0~100%)

uint32_t **polarity_active_low**

Specify the polarity of carrier, by default it' s modulated to base signal' s high level

uint32_t **always_on**

If set, the carrier can always exist even there' s not transfer undergoing

struct *rmt_carrier_config_t*::[anonymous] **flags**

Carrier config flags

Header File

- components/driver/include/driver/rmt_encoder.h

Functions

esp_err_t **rmt_new_bytes_encoder**(const *rmt_bytes_encoder_config_t* *config, *rmt_encoder_handle_t* *ret_encoder)

Create RMT bytes encoder, which can encode byte stream into RMT symbols.

Parameters

- **config** –**[in]** Bytes encoder configuration
- **ret_encoder** –**[out]** Returned encoder handle

Returns

- ESP_OK: Create RMT bytes encoder successfully
- ESP_ERR_INVALID_ARG: Create RMT bytes encoder failed because of invalid argument
- ESP_ERR_NO_MEM: Create RMT bytes encoder failed because out of memory
- ESP_FAIL: Create RMT bytes encoder failed because of other error

esp_err_t **rmt_bytes_encoder_update_config**(*rmt_encoder_handle_t* bytes_encoder, const *rmt_bytes_encoder_config_t* *config)

Update the configuration of the bytes encoder.

Note: The configurations of the bytes encoder is also set up by `rmt_new_bytes_encoder()`. This function is used to update the configuration of the bytes encoder at runtime.

Parameters

- **bytes_encoder** –**[in]** Bytes encoder handle, created by e.g `rmt_new_bytes_encoder()`
- **config** –**[in]** Bytes encoder configuration

Returns

- ESP_OK: Update RMT bytes encoder successfully
- ESP_ERR_INVALID_ARG: Update RMT bytes encoder failed because of invalid argument
- ESP_FAIL: Update RMT bytes encoder failed because of other error

esp_err_t **rmt_new_copy_encoder**(const *rmt_copy_encoder_config_t* *config, *rmt_encoder_handle_t* *ret_encoder)

Create RMT copy encoder, which copies the given RMT symbols into RMT memory.

Parameters

- **config** –**[in]** Copy encoder configuration
- **ret_encoder** –**[out]** Returned encoder handle

Returns

- ESP_OK: Create RMT copy encoder successfully

- ESP_ERR_INVALID_ARG: Create RMT copy encoder failed because of invalid argument
- ESP_ERR_NO_MEM: Create RMT copy encoder failed because out of memory
- ESP_FAIL: Create RMT copy encoder failed because of other error

esp_err_t **rmt_del_encoder**(*rmt_encoder_handle_t* encoder)

Delete RMT encoder.

Parameters **encoder** –**[in]** RMT encoder handle, created by e.g `rmt_new_bytes_encoder()`

Returns

- ESP_OK: Delete RMT encoder successfully
- ESP_ERR_INVALID_ARG: Delete RMT encoder failed because of invalid argument
- ESP_FAIL: Delete RMT encoder failed because of other error

esp_err_t **rmt_encoder_reset**(*rmt_encoder_handle_t* encoder)

Reset RMT encoder.

Parameters **encoder** –**[in]** RMT encoder handle, created by e.g `rmt_new_bytes_encoder()`

Returns

- ESP_OK: Reset RMT encoder successfully
- ESP_ERR_INVALID_ARG: Reset RMT encoder failed because of invalid argument
- ESP_FAIL: Reset RMT encoder failed because of other error

void ***rmt_alloc_encoder_mem**(size_t size)

A helper function to allocate a proper memory for RMT encoder.

Parameters **size** –Size of memory to be allocated

Returns Pointer to the allocated memory if the allocation is successful, NULL otherwise

Structures

struct **rmt_encoder_t**

Interface of RMT encoder.

Public Members

size_t (***encode**)(*rmt_encoder_t* *encoder, *rmt_channel_handle_t* tx_channel, const void *primary_data, size_t data_size, *rmt_encode_state_t* *ret_state)

Encode the user data into RMT symbols and write into RMT memory.

Note: The encoding function will also be called from an ISR context, thus the function must not call any blocking API.

Note: It's recommended to put this function implementation in the IRAM, to achieve a high performance and less interrupt latency.

Param encoder **[in]** Encoder handle

Param tx_channel **[in]** RMT TX channel handle, returned from `rmt_new_tx_channel()`

Param primary_data **[in]** App data to be encoded into RMT symbols

Param data_size **[in]** Size of primary_data, in bytes

Param ret_state **[out]** Returned current encoder's state

Return Number of RMT symbols that the primary data has been encoded into

esp_err_t (***reset**)(*rmt_encoder_t* *encoder)

Reset encoding state.

Param encoder **[in]** Encoder handle
Return
- ESP_OK: reset encoder successfully
- ESP_FAIL: reset encoder failed

esp_err_t (***del**)(*rmt_encoder_t* *encoder)

Delete encoder object.

Param encoder **[in]** Encoder handle
Return
- ESP_OK: delete encoder successfully
- ESP_FAIL: delete encoder failed

struct **rmt_bytes_encoder_config_t**

Bytes encoder configuration.

Public Members

rmt_symbol_word_t **bit0**

How to represent BIT0 in RMT symbol

rmt_symbol_word_t **bit1**

How to represent BIT1 in RMT symbol

uint32_t **msb_first**

Whether to encode MSB bit first

struct *rmt_bytes_encoder_config_t*::[anonymous] **flags**

Encoder config flag

struct **rmt_copy_encoder_config_t**

Copy encoder configuration.

Type Definitions

typedef struct *rmt_encoder_t* **rmt_encoder_t**

Type of RMT encoder.

Enumerations

enum **rmt_encode_state_t**

RMT encoding state.

Values:

enumerator **RMT_ENCODING_RESET**

The encoding session is in reset state

enumerator **RMT_ENCODING_COMPLETE**

The encoding session is finished, the caller can continue with subsequent encoding

enumerator **RMT_ENCODING_MEM_FULL**

The encoding artifact memory is full, the caller should return from current encoding session

Header File

- components/driver/include/driver/rmt_types.h

Structures

struct **rmt_tx_done_event_data_t**

Type of RMT TX done event data.

Public Members

size_t **num_symbols**

The number of transmitted RMT symbols (only one round is counted if it's a loop transmission)

struct **rmt_rx_done_event_data_t**

Type of RMT RX done event data.

Public Members

rmt_symbol_word_t ***received_symbols**

Point to the received RMT symbols

size_t **num_symbols**

The number of received RMT symbols

Type Definitions

typedef struct rmt_channel_t ***rmt_channel_handle_t**

Type of RMT channel handle.

typedef struct rmt_sync_manager_t ***rmt_sync_manager_handle_t**

Type of RMT synchronization manager handle.

typedef struct *rmt_encoder_t* ***rmt_encoder_handle_t**

Type of RMT encoder handle.

typedef bool (***rmt_tx_done_callback_t**)(*rmt_channel_handle_t* tx_chan, const *rmt_tx_done_event_data_t* *edata, void *user_ctx)

Prototype of RMT event callback.

Param tx_chan [in] RMT channel handle, created from `rmt_new_tx_channel()`

Param edata [in] Point to RMT event data. The lifecycle of this pointer memory is inside this function, user should copy it into static memory if used outside this funcion.

Param user_ctx [in] User registered context, passed from `rmt_tx_register_event_callbacks()`

Return Whether a high priority task has been waken up by this callback function

typedef bool (***rmt_rx_done_callback_t**)(*rmt_channel_handle_t* rx_chan, const *rmt_rx_done_event_data_t* *edata, void *user_ctx)

Prototype of RMT event callback.

Param rx_chan [in] RMT channel handle, created from `rmt_new_rx_channel()`

Param edata [in] Point to RMT event data. The lifecycle of this pointer memory is inside this function, user should copy it into static memory if used outside this funcion.

Param user_ctx [in] User registered context, passed from `rmt_rx_register_event_callbacks()`

Return Whether a high priority task has been waken up by this function

Header File

- components/hal/include/hal/rmt_types.h

Unions

union **rmt_symbol_word_t**

#include <rmt_types.h> The layout of RMT symbol stored in memory, which is decided by the hardware design.

Public Members

uint16_t **duration0**

Duration of level0

uint16_t **level0**

Level of the first part

uint16_t **duration1**

Duration of level1

uint16_t **level1**

Level of the second part

struct *rmt_symbol_word_t*::[anonymous] **[anonymous]**

uint32_t **val**

Equivalent unsigned value for the RMT symbol

Type Definitions

typedef *soc_periph_rmt_clk_src_t* **rmt_clock_source_t**

RMT group clock source.

Note: User should select the clock source based on the power and resolution requirement

2.6.15 SD Pull-up Requirements

Espressif hardware products are designed for multiple use cases which may require different pull states on pins. For this reason, the pull state of particular pins on certain products will need to be adjusted to provide the pull-ups required in the SD bus.

SD pull-up requirements apply to cases where ESP32 uses the SPI or SDMMC controller to communicate with SD cards. When an SD card is operating in SPI mode or 1-bit SD mode, the CMD and DATA (DAT0 - DAT3) lines of the SD bus must be pulled up by 10 kOhm resistors. SD cards and SDIO devices should also have pull-ups on all above-mentioned lines (regardless of whether these lines are connected to the host) in order to prevent them from entering a wrong state.

By default, the MTDI bootstrapping pin is incompatible with the DAT2 line pull-up if the flash voltage is 3.3 V. For more information, see *MTDI Strapping Pin* below.

This document has the following structure:

- *Overview of compatibility* between the default pull states on pins of Espressif ' s products and the states required by the SD bus
- *Solutions* - ideas on how to resolve compatibility issues
- *Related information* - other relevant information

Overview of Compatibility

This section provides an overview of compatibility issues that might occur when using SDIO (secure digital input output). Since the SD bus needs to be connected to pull-ups, these issues should be resolved regardless of whether they are related to master (host) or slave (device). Each issue has links to its respective solution. A solution for a host and device may differ.

Systems on a Chip (SoCs)

- ESP32 (except for D2WD versions, see ESP32 datasheet):
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2* for models with 3.3 V flash chip
- ESP32-D2WD:
  - *No Pull-ups*
  - *No Pull-up on GPIO12*

Systems in Packages (SIP)

- ESP32-PICO-D4:
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2*

Modules

- ESP32-WROOM-32 Series, including ESP32-WROOM-32, ESP32-WROOM-32D, ESP32-WROOM-32U, and ESP32-SOLO-1
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2*
- ESP32-WROVER Series, including ESP32-WROVER and ESP32-WROVER-I
  - *No Pull-ups*
- ESP32-WROVER-B Series, including ESP32-WROVER-B and ESP32-WROVER-IB
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2*

Development Boards

- ESP32-PICO-KIT, including PICO-KIT v4.1, v4.0, and v3
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2*
  - *Download Mode Not Working (minor issue)*
- ESP32-DevKitC, including ESP32-DevKitC v4 and v2
  - *No Pull-ups*
  - *Conflicts Between Bootstrap and SDIO on DAT2*
  - *Download Mode Not Working (minor issue)*
- ESP-WROVER-KIT
  - Required pull-ups are provided
  - *Pull-up Conflicts on GPIO13* (v4.1, v3, v2, and v1)
  - *Conflicts Between Bootstrap and SDIO on DAT2* (v4.1, v2, and v1)
  - *Download Mode Not Working (minor issue)* (v2, v1)

  You can determine the version of your ESP23-WROVER-KIT by checking which module is mounted on it:
  - ESP32-WROVER-B on v4.1
  - ESP32-WROVER on v3
  - ESP32-WROOM-32 on v1 and v2
- ESP32-LyraTD-MSC
  - Required pull-ups are provided
  - *Conflicts Between Bootstrap and SDIO on DAT2*
- ESP32-LyraT
  - Required pull-ups are provided
  - *Pull-up Conflicts on GPIO13*

Non-Espressif Hosts Please make sure that your SDIO host provides necessary pull-ups for all SD bus signals.

Solutions

No Pull-ups If you use a development board without pull-ups, you can do the following:

- If your host and slave device are on separate boards, replace one of them with a board that has pull-ups. For the list of Espressif's development boards with pull-ups, go to *Development Boards*.
- Attach external pull-ups by connecting each pin which requires a pull-up to VDD via a 10 kOhm resistor.

Pull-up Conflicts on GPIO13 If DAT3 of your device is not properly pulled up, you have the following options:

- Use 1-bit SD mode and tie the device's DAT3 to VDD
- Use SPI mode
- **Perform one of the following actions on the GPIO13 pin:**
  - Remove the pull-down resistors
  - Attach a pull-up resistor of less than 5 kOhm (2 kOhm suggested)
  - Pull it up or drive it high either by using the host or with 3.3 V on VDD in 1-bit SD mode

Conflicts Between Bootstrap and SDIO on DAT2 There is a conflict between the boot strapping requirements of the ESP32 and the SDIO protocol. For details, see *MTDI Strapping Pin*.

To resolve the conflict, you have the following options:

1. (Recommended) Burn the flash voltage selection eFuses. This will permanently configure the internal regulator's output voltage to 3.3 V, and GPIO12 will not be used as a bootstrapping pin. After that, connect a pull-up resistor to GPIO12.

Warning: Burning eFuses is irreversible! The issue list above might be out of date, so please make sure that the module you are burning has a 3.3 V flash chip by checking the information on https://www.espressif.com/. If you burn the 3.3 V eFuses on a module with a 1.8 V flash chip, the module will stop functioning.

If you are sure that you need to irreversibly burn eFuses, go to your ESP-IDF directory and run the following command:

```
components/esptool_py/esptool/espefuse.py set_flash_voltage 3.3V
```

This command will burn the *XPD_SDIO_TIEH*, *XPD_SDIO_FORCE*, and *XPD_SDIO_REG* eFuses. After all the three eFuses are burned to value 1, the internal VDD_SDIO flash voltage regulator will be permanently set to 3.3 V. You will see the following log if the burning succeeds:

```
espefuse.py v2.6
Connecting....

Enable internal flash voltage regulator (VDD_SDIO) to 3.3 V.
The following eFuses are burned: XPD_SDIO_FORCE, XPD_SDIO_REG, XPD_SDIO_TIEH.
This is an irreversible operation.
Type 'BURN' (all capitals) to continue.
BURN
VDD_SDIO setting complete.
```

To check the status of the eFuses, run:

```
``components/esptool_py/esptool/espefuse.py summary``
```

If running from an automated flashing script, `espefuse.py` has an option `--do-not-confirm`.

For more details, see *ESP32 Technical Reference Manual* [PDF].

2. **If using 1-bit SD mode or SPI mode**, disconnect the DAT2 pin and make sure it is pulled high. For this, do one the following:
   - Leave the host's DAT2 floating and directly connect the slave's DAT2 to VDD.
   - For a slave device, build a firmware with the option `SDIO_SLAVE_FLAG_DAT2_DISABLED` and reflash your device. This option will help avoid slave detecting on the DAT2 line. Note that 4-bit SD mode will no longer be supported by the standard Card Common Control Register (CCCR); however, the host will not be aware of that. The use of 4-bit SD mode will have to be disabled on the host's side.

No Pull-up on GPIO12 Your module is compatible with the SDIO protocol. Just connect GPIO12 to VDD via a 10 kOhm resistor.

Download Mode Not Working (minor issue) When the GPIO2 pin is pulled high in accordance with the SD pull-up requirements, you cannot enter Download mode because GPIO2 is a bootstrapping pin which in this case must be pulled low.

There are the following solutions:

- For boards that require shorting the GPIO0 and GPIO2 pins with a jumper, put the jumper in place, and the auto-reset circuit will pull GPIO2 low along with GPIO0 before entering Download mode.
- For boards with components attached to their GPIO2 pin (such as pull-down resistors and/or LEDs), check the schematic of your development board for anything connected to GPIO2.
  - **LEDs** would not affect operation in most cases.
  - **Pull-down resistors** can interfere with DAT0 signals and must be removed.

If the above solutions do not work for you, please determine if it is the host or slave device that has pull-ups affecting their GPIO2, then locate these pull-ups and remove them.

Related Information

MTDI Strapping Pin MTDI (GPIO12) is used as a bootstrapping pin to select the output voltage of an internal regulator (VDD_SDIO) which powers the flash chip. This pin has an internal pull-down, so, if left unconnected, it will read low level at startup, which will lead to selecting the default 3.3 V operation.

All ESP32-WROVER modules, excluding ESP32-WROVER-B, use 1.8 V flash and have internal pull-ups on GPIO12. Other modules that use 3.3 V flash have no pull-ups on the GPIO12 pin, and this pin is slightly pulled down internally.

When adding a pull-up to this pin for SD card operation, consider the following:

- For boards that do not use the internal regulator (VDD_SDIO) to power flash, GPIO12 can be pulled high.
- For boards using 1.8 V flash chips, GPIO12 needs to be pulled high at reset. This is fully compatible with the SD card operation.
- On boards using the internal regulator and a 3.3 V flash chip, GPIO12 must be pulled low at reset. This is incompatible with the SD card operation. For reference information on compatibility of Espressif's boards with the SD card operation, see *Overview of Compatibility*.

Internal Pull-ups and Strapping Requirements Using external resistors is always preferable. However, Espressif's products have internal weak pull-up and pull-down resistors which can be enabled and used instead of external ones. Please keep in mind that this solution cannot guarantee reliable SDIO communication.

With that said, the information about these internal pull-ups and strapping requirements can still be useful. Espressif hardware products have different weak internal pull-ups / pull-downs connected to CMD and DATA pins. The table below shows the default pull-up and pull-down states of the CMD and DATA pins.

The following abbreviations are used in the table:

- **WPU**: Weak pull-up inside the SoC
- **WPD**: Weak pull-down inside the SoC
- **PU**: Pull-up inside Espressif modules but outside the SoC

Table 6: Default pull-up and pull-down states of the CMD and DATA pins

GPIO number	Pin Name	Startup State	Strapping Requirement
15	CMD	WPU	
2	DAT0	WPD	Low for Download mode
4	DAT1	WPD	
12	DAT2	PU for 1.8 V flash; WPD for 3.3 V flash	High for 1.8 V flash; Low for 3.3 V flash
13	DAT3	WPU	

2.6.16 SDMMC Host Driver

Overview

ESP32's SDMMC host peripheral has two slots. Each slot can be used independently to connect to an SD card, SDIO device or eMMC chip.

- Slot 0 (*SDMMC_HOST_SLOT_0*) is an 8-bit slot. It uses `HS1_*` signals in the PIN MUX.
- Slot 1 (*SDMMC_HOST_SLOT_1*) is a 4-bit slot. It uses `HS2_*` signals in the PIN MUX.

The slots are connected to ESP32 GPIOs using IO MUX. Pin mappings of these slots are given in the table below.

Signal	Slot 0	Slot 1
CMD	GPIO11	GPIO15
CLK	GPIO6	GPIO14
D0	GPIO7	GPIO2
D1	GPIO8	GPIO4
D2	GPIO9	GPIO12
D3	GPIO10	GPIO13
D4	GPIO16	
D5	GPIO17	
D6	GPIO5	
D7	GPIO18	
CD	any input via GPIO matrix	
WP	any input via GPIO matrix	

The Card Detect and Write Protect signals can be routed to arbitrary pins using the GPIO matrix. To reserve the pins, set the `cd` and `wp` members of the *`sdmmc_slot_config_t`* structure before calling *`sdmmc_host_init_slot()`*. Please note that it is not advised to specify a Card Detect pin when working with SDIO cards, because the card detect signal in ESP32 can also trigger SDIO slave interrupt.

Warning: Pins used by Slot 0 (`HS1_*`) are also used to connect the SPI flash chip in ESP32-WROOM and ESP32-WROVER modules. These pins cannot be shared between an SD card and SPI flash. If you need to use Slot 0, connect SPI flash to different pins and set eFuses accordingly.

Supported Speed Modes

SDMMC Host driver supports the following speed modes:

- Default Speed (20 MHz), 1/4-line (with SD cards), and 1/4/8-line (with 3.3 V eMMC)
- High Speed (40 MHz), 1/4-line (with SD cards), and 1/4/8-line (with 3.3 V eMMC)
- High Speed DDR (40 MHz), 4-line (with 3.3 V eMMC)

Speed modes not supported at present:

- High Speed DDR mode, 8-line eMMC
- UHS-I 1.8 V modes, 4-line SD cards

Using the SDMMC Host Driver

Of all the functions listed below, only the following ones will be used directly by most applications:

- *`sdmmc_host_init()`*
- *`sdmmc_host_init_slot()`*
- *`sdmmc_host_deinit()`*

Other functions, such as the ones given below, will be called by the SD/MMC protocol layer via function pointers in the *`sdmmc_host_t`* structure:

- *`sdmmc_host_set_bus_width()`*
- *`sdmmc_host_set_card_clk()`*
- *`sdmmc_host_do_transaction()`*

Configuring Bus Width and Frequency

With the default initializers for *`sdmmc_host_t`* and *`sdmmc_slot_config_t`* (*`SDMMC_HOST_DEFAULT`* and *`SDMMC_SLOT_CONFIG_DEFAULT`*), SDMMC Host driver will attempt to use the widest bus supported by the card (4 lines for SD, 8 lines for eMMC) and the frequency of 20 MHz.

In the designs where communication at 40 MHz frequency can be achieved, it is possible to increase the bus frequency by changing the `max_freq_khz` field of *sdmmc_host_t*:

```
sdmmc_host_t host = SDMMC_HOST_DEFAULT();
host.max_freq_khz = SDMMC_FREQ_HIGHSPEED;
```

To configure the bus width, set the `width` field of *sdmmc_slot_config_t*. For example, to set 1-line mode:

```
sdmmc_slot_config_t slot = SDMMC_SLOT_CONFIG_DEFAULT();
slot.width = 1;
```

DDR Mode for eMMC chips

By default, DDR mode will be used if:

- SDMMC host frequency is set to *SDMMC_FREQ_HIGHSPEED* in *sdmmc_host_t* structure, and
- eMMC chip reports DDR mode support in its CSD register

DDR mode places higher requirements for signal integrity. To disable DDR mode while keeping *SDMMC_FREQ_HIGHSPEED* frequency, clear *SDMMC_HOST_FLAG_DDR* bit in `flags` field of *sdmmc_host_t*:

```
sdmmc_host_t host = SDMMC_HOST_DEFAULT();
host.max_freq_khz = SDMMC_FREQ_HIGHSPEED;
host.flags &= ~SDMMC_HOST_FLAG_DDR;
```

See also

See *SD/SDIO/MMC Driver* for the higher level driver which implements the protocol layer.

See *SD SPI Host Driver* for a similar driver which uses the SPI controller and is limited to SD protocol's SPI mode.

See *SD Pull-up Requirements* for pullup support and compatibilities of modules and development kits.

API Reference

Header File

- components/driver/include/driver/sdmmc_host.h

Functions

esp_err_t **sdmmc_host_init**(void)

Initialize SDMMC host peripheral.

Note: This function is not thread safe

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdmmc_host_init was already called
- ESP_ERR_NO_MEM if memory can not be allocated

esp_err_t **sdmmc_host_init_slot**(int slot, const *sdmmc_slot_config_t* *slot_config)

Initialize given slot of SDMMC peripheral.

On the ESP32, SDMMC peripheral has two slots:

- Slot 0: 8-bit wide, maps to HS1_* signals in PIN MUX
- Slot 1: 4-bit wide, maps to HS2_* signals in PIN MUX

Card detect and write protect signals can be routed to arbitrary GPIOs using GPIO matrix.

Note: This function is not thread safe

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **slot_config** –additional configuration for the slot

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if host has not been initialized using sdmmc_host_init

esp_err_t **sdmmc_host_set_bus_width** (int slot, size_t width)

Select bus width to be used for data transfer.

SD/MMC card must be initialized prior to this command, and a command to set bus width has to be sent to the card (e.g. SD_APP_SET_BUS_WIDTH)

Note: This function is not thread safe

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **width** –bus width (1, 4, or 8 for slot 0; 1 or 4 for slot 1)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if slot number or width is not valid

size_t **sdmmc_host_get_slot_width** (int slot)

Get bus width configured in `sdmmc_host_init_slot` to be used for data transfer.

Parameters **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
Returns configured bus width of the specified slot.

esp_err_t **sdmmc_host_set_card_clk** (int slot, uint32_t freq_khz)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note: This function is not thread safe

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **freq_khz** –card clock frequency, in kHz

Returns
- ESP_OK on success
- other error codes may be returned in the future

esp_err_t **sdmmc_host_set_bus_ddr_mode** (int slot, bool ddr_enabled)

Enable or disable DDR mode of SD interface.

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **ddr_enabled** –enable or disable DDR mode

Returns
- ESP_OK on success

- ESP_ERR_NOT_SUPPORTED if DDR mode is not supported on this slot

esp_err_t **sdmmc_host_set_cclk_always_on**(int slot, bool cclk_always_on)

Enable or disable always-on card clock When cclk_always_on is false, the host controller is allowed to shut down the card clock between the commands. When cclk_always_on is true, the clock is generated even if no command is in progress.

Parameters
- **slot** –slot number
- **cclk_always_on** –enable or disable always-on clock

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the slot number is invalid

esp_err_t **sdmmc_host_do_transaction**(int slot, *sdmmc_command_t* *cmdinfo)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Attention Data buffer passed in cmdinfo->data must be in DMA capable memory

---

Note: This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdmmc_host_do_transaction as long as other sdmmc_host_* functions are not called.

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **cmdinfo** –pointer to structure describing command and data to transfer

Returns
- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response
- ESP_ERR_INVALID_SIZE if the size of data transfer is not valid in SD protocol
- ESP_ERR_INVALID_ARG if the data buffer is not in DMA capable memory

esp_err_t **sdmmc_host_io_int_enable**(int slot)

Enable IO interrupts.

This function configures the host to accept SDIO interrupts.

Parameters **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
Returns returns ESP_OK, other errors possible in the future

esp_err_t **sdmmc_host_io_int_wait**(int slot, TickType_t timeout_ticks)

Block until an SDIO interrupt is received, or timeout occurs.

Parameters
- **slot** –slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- **timeout_ticks** –number of RTOS ticks to wait for the interrupt

Returns
- ESP_OK on success (interrupt received)
- ESP_ERR_TIMEOUT if the interrupt did not occur within timeout_ticks

esp_err_t **sdmmc_host_deinit**(void)

Disable SDMMC host and release allocated resources.

Note: This function is not thread safe

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdmmc_host_init function has not been called

Structures

struct **sdmmc_slot_config_t**

Extra configuration for SDMMC peripheral slot

Public Members

gpio_num_t **gpio_cd**

GPIO number of card detect signal.

gpio_num_t **cd**

GPIO number of card detect signal; shorter name.

gpio_num_t **gpio_wp**

GPIO number of write protect signal.

gpio_num_t **wp**

GPIO number of write protect signal; shorter name.

uint8_t **width**

Bus width used by the slot (might be less than the max width supported)

uint32_t **flags**

Features used by this slot.

Macros

SDMMC_HOST_SLOT_0

SDMMC slot 0.

SDMMC_HOST_SLOT_1

SDMMC slot 1.

SDMMC_HOST_DEFAULT()

Default *sdmmc_host_t* structure initializer for SDMMC peripheral.

Uses SDMMC peripheral, with 4-bit mode enabled, and max frequency set to 20MHz

SDMMC_SLOT_FLAG_INTERNAL_PULLUP

Enable internal pullups on enabled pins. The internal pullups are insufficient however, please make sure external pullups are connected on the bus. This is for debug / example purpose only.

SDMMC_SLOT_NO_CD

indicates that card detect line is not used

SDMMC_SLOT_NO_WP

indicates that write protect line is not used

SDMMC_SLOT_WIDTH_DEFAULT

use the maximum possible width for the slot

SDMMC_SLOT_CONFIG_DEFAULT()

Macro defining default configuration of SDMMC host slot

2.6.17 SD SPI Host Driver

Overview

The SD SPI host driver allows communicating with one or more SD cards by the SPI Master driver which makes use of the SPI host. Each card is accessed through an SD SPI device represented by an *sdspi_dev_handle_t* spi_handle returned when attaching the device to an SPI bus by calling *sdspi_host_init_device*. The bus should be already initialized before (by *spi_bus_initialize*).

This driver's naming pattern was adopted from the *SDMMC Host* driver due to their similarity. Likewise, the APIs of both drivers are also very similar.

SD SPI driver (access the SD card in SPI mode) offers lower throughput but makes pin selection more flexible. With the help of the GPIO matrix, an SPI peripheral's signals can be routed to any ESP32 pin. Otherwise, if SDMMC host driver is used (See *SDMMC Host*) to access the card in SD 1-bit/4-bit mode, higher throughput can be reached but it requires routing the signals through their dedicated IO_MUX pins only.

With the help of *SPI Master driver* based on, the SPI bus can be shared among SD cards and other SPI devices. The SPI Master driver will handle exclusive access from different tasks.

The SD SPI driver uses software-controlled CS signal.

How to Use

Firstly, use the macro *SDSPI_DEVICE_CONFIG_DEFAULT* to initialize a structure *sdmmc_slot_config_t*, which is used to initialize an SD SPI device. This macro will also fill in the default pin mappings, which is same as the pin mappings of SDMMC host driver. Modify the host and pins of the structure to desired value. Then call *sdspi_host_init_device* to initialize the SD SPI device and attach to its bus.

Then use *SDSPI_HOST_DEFAULT* macro to initialize a *sdmmc_host_t* structure, which is used to store the state and configurations of upper layer (SD/SDIO/MMC driver). Modify the *slot* parameter of the structure to the SD SPI device spi_handle just returned from *sdspi_host_init_device*. Call *sdmmc_card_init* with the *sdmmc_host_t* to probe and initialize the SD card.

Now you can use SD/SDIO/MMC driver functions to access your card!

Other Details

Only the following driver's API functions are normally used by most applications:

- *sdspi_host_init()*
- *sdspi_host_init_device()*
- *sdspi_host_remove_device()*
- *sdspi_host_deinit()*

Other functions are mostly used by the protocol level SD/SDIO/MMC driver via function pointers in the *sdmmc_host_t* structure. For more details, see *the SD/SDIO/MMC Driver*.

Note: SD over SPI does not support speeds above *SDMMC_FREQ_DEFAULT* due to the limitations of the SPI driver.

Warning: If you want to share the SPI bus among SD card and other SPI devices, there are some restrictions, see *Sharing the SPI bus among SD card and other SPI devices*.

Related Docs

Sharing the SPI bus among SD card and other SPI devices The SD card has a SPI mode, which allows it to be communicated to as a SPI device. But there are some restrictions that we need to pay attention to.

Pin loading of other devices When adding more devices onto the same bus, the overall pin loading increases. The loading consists of AC loading (pin capacitor) and DC loading (pull-ups).

AC loading SD cards, which are designed for high-speed communications, have small pin capacitors (AC loading) to work until 50MHz. However, the other attached devices will increase the pin's AC loading.

Heavy AC loading of a pin may prevent the pin from being toggled quickly. By using an oscilloscope, you will see the edges of the pin become smoother and not ideal any more (the gradient of the edge is smaller). The setup timing requirements of an SD card may be violoated when the card is connected to such bus. Even worse, the clock from the host may not be recognized by the SD card and other SPI devices on the same bus.

This issue may be more obvious if other attached devices are not designed to work at the same frequency as the SD card, because they may have larger pin capacitors.

To see if your pin AC loading is too heavy, you can try the following tests:

(Terminology: **launch edge**: at which clock edge the data start to toggle; **latch edge**: at which clock edge the data is supposed to be sampled by the receiver, for SD cad, it's the rising edge.)

1. Use an oscilloscope to see the clock and compare the data line to the clock. - If you see the clock is not fast enough (for example, the rising/falling edge is longer than 1/4 of the clock cycle), it means the clock is skewed too much. - If you see the data line unstable before the latch edge of the clock, it means the load of the data line is too large.
   You may also observed the corresponding phenomenon (data delayed largely from launching edge of clock) with logic analyzers. But it's not as obvious as with an oscilloscope.
2. Try to use slower clock frequency.
   If the lower frequency can work while the higher frequency can't, it's an indication of the AC loading on the pins is too large.

If the AC loading of the pins is too large, you can either use other faster devices (with lower pin load) or slow down the clock speed.

DC loading The pull-ups required by SD cards are usually around 10 kOhm to 50 kOhm, which may be too strong for some other SPI devices.

Check the specification of your device about its DC output current , it should be larger than 700uA, otherwise the device output may not be read correctly.

Initialization sequence

Note: If you see any problem in the following steps, please make sure the timing is correct first. You can try to slow down the clock speed (SDMMC_FREQ_PROBING = 400 KHz for SD card) to avoid the influence of pin AC loading (see above section).

When using ab SD card with other SPI devices on the same SPI bus, due to the restrictions of the SD card startup flow, the following initialization sequence should be followed: (See also storage/sd_card)

1. Initialize the SPI bus properly by *spi_bus_initialize*.

2. Tie the CS lines of all other devices than the SD card to high. This is to avoid conflicts to the SD card in the following step.
   You can do this by either:
   1. Attach devices to the SPI bus by calling *spi_bus_add_device*. This function will initialize the GPIO that is used as CS to the idle level: high.
   2. Initialize GPIO on the CS pin that needs to be tied up before actually adding a new device.
   3. Rely on the internal/external pull-up (not recommended) to pull-up all the CS pins when the GPIOs of ESP are not initialized yet. You need to check carefull the pull-up is strong enough and there are no other pull-downs that will influence the pull-up (For example, internal pull-down should be enabled).
3. Mount the card to the filesystem by calling *esp_vfs_fat_sdspi_mount*.
   This step will put the SD card into the SPI mode, which SHOULD be done before all other SPI communications on the same bus. Otherwise the card will stay in the SD mode, in which mode it may randomly respond to any SPI communications on the bus, even when its CS line is not addressed.
   If you want to test this behavior, please also note that, once the card is put into SPI mode, it will not return to SD mode before next power cycle, i.e. powered down and powered up again.
4. Now you can talk to other SPI devices freely!

API Reference

Header File

- components/driver/include/driver/sdspi_host.h

Functions

esp_err_t **sdspi_host_init**(void)

Initialize SD SPI driver.

Note: This function is not thread safe

Returns

- ESP_OK on success
- other error codes may be returned in future versions

esp_err_t **sdspi_host_init_device**(const *sdspi_device_config_t* *dev_config, *sdspi_dev_handle_t* *out_handle)

Attach and initialize an SD SPI device on the specific SPI bus.

Note: This function is not thread safe

Note: Initialize the SPI bus by `spi_bus_initialize()` before calling this function.

Note: The SDIO over sdspi needs an extra interrupt line. Call `gpio_install_isr_service()` before this function.

Parameters

- **dev_config** –pointer to device configuration structure
- **out_handle** –Output of the handle to the sdspi device.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if sdspi_host_init_device has invalid arguments
- ESP_ERR_NO_MEM if memory can not be allocated

- other errors from the underlying spi_master and gpio drivers

esp_err_t **sdspi_host_remove_device**(*sdspi_dev_handle_t* handle)

Remove an SD SPI device.

Parameters **handle** –Handle of the SD SPI device
Returns Always ESP_OK

esp_err_t **sdspi_host_do_transaction**(*sdspi_dev_handle_t* handle, *sdmmc_command_t* *cmdinfo)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note: This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdspi_host_do_transaction as long as other sdspi_host_* functions are not called.

Parameters
- **handle** –Handle of the sdspi device
- **cmdinfo** –pointer to structure describing command and data to transfer

Returns
- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response

esp_err_t **sdspi_host_set_card_clk**(*sdspi_dev_handle_t* host, uint32_t freq_khz)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note: This function is not thread safe

Parameters
- **host** –Handle of the sdspi device
- **freq_khz** –card clock frequency, in kHz

Returns
- ESP_OK on success
- other error codes may be returned in the future

esp_err_t **sdspi_host_deinit**(void)

Release resources allocated using sdspi_host_init.

Note: This function is not thread safe

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdspi_host_init function has not been called

esp_err_t **sdspi_host_io_int_enable**(*sdspi_dev_handle_t* handle)

Enable SDIO interrupt.

Parameters **handle** –Handle of the sdspi device
Returns

- ESP_OK on success

esp_err_t **sdspi_host_io_int_wait** (*sdspi_dev_handle_t* handle, TickType_t timeout_ticks)

Wait for SDIO interrupt until timeout.

Parameters

- **handle** –Handle of the sdspi device
- **timeout_ticks** –Ticks to wait before timeout.

Returns

- ESP_OK on success

Structures

struct **sdspi_device_config_t**

Extra configuration for SD SPI device.

Public Members

spi_host_device_t **host_id**

SPI host to use, SPIx_HOST (see spi_types.h).

gpio_num_t **gpio_cs**

GPIO number of CS signal.

gpio_num_t **gpio_cd**

GPIO number of card detect signal.

gpio_num_t **gpio_wp**

GPIO number of write protect signal.

gpio_num_t **gpio_int**

GPIO number of interrupt line (input) for SDIO card.

Macros

SDSPI_DEFAULT_HOST

SDSPI_DEFAULT_DMA

SDSPI_HOST_DEFAULT()

Default *sdmmc_host_t* structure initializer for SD over SPI driver.

Uses SPI mode and max frequency set to 20MHz

'slot' should be set to an sdspi device initialized by `sdspi_host_init_device()`.

SDSPI_SLOT_NO_CS

indicates that card select line is not used

SDSPI_SLOT_NO_CD

indicates that card detect line is not used

SDSPI_SLOT_NO_WP

indicates that write protect line is not used

SDSPI_SLOT_NO_INT

indicates that interrupt line is not used

SDSPI_DEVICE_CONFIG_DEFAULT()

Macro defining default configuration of SD SPI device.

Type Definitions

typedef int **sdspi_dev_handle_t**

Handle representing an SD SPI device.

2.6.18 SDIO Card Slave Driver

Overview

The ESP32 SDIO Card peripherals (Host, Slave) shares two sets of pins as below table. The first set is usually occupied by SPI0 bus which is responsible for the SPI flash holding the code to run. This means SDIO slave driver can only runs on the second set of pins while SDIO host is not using it.

The SDIO slave can run under 3 modes: SPI, 1-bit SD and 4-bit SD modes, which is detected automatically by the hardware. According to the SDIO specification, CMD and DAT0-3 lines should be pulled up no matter in 1-bit, 4-bit or SPI mode.

Connections

Pin Name	Corresponding pins in SPI mode	Slot1	Slot2
		GPIO Number	
CLK	SCLK	6	14
CMD	MOSI	11	15
DAT0	MISO	7	2
DAT1	Interrupt	8	4
DAT2	N.C. (pullup)	9	12
DAT3	#CS	10	13

- 1-bit SD mode: Connect CLK, CMD, DAT0, DAT1 pins and the ground.
- 4-bit SD mode: Connect all pins and the ground.
- SPI mode: Connect SCLK, MOSI, MISO, Interrupt, #CS pins and the ground.

Note: Please check if CMD and DATA lines D0-D3 of the card are properly pulled up by 10 KOhm resistors. This should be ensured even in 1-bit mode or SPI mode. Most official modules don't offer these pullups internally. If you are using official development boards, check *Overview of Compatibility* to see whether your development boards have such pullups.

Note: Most official modules have conflicts on strapping pins with the SDIO slave function. If you are using a ESP32 module with 3.3 V flash inside, you have to burn the EFUSE when you are developing on the module for the first time. See *Overview of Compatibility* to see how to make your modules compatible with the SDIO.

Here is a list for modules/kits with 3.3 V flash:

- Modules: ESP32-PICO-D4, ESP32-WROOM-32 series (including ESP32-SOLO-1), ESP32-WROVER-B and ESP32-WROVER-IB

- Kits: ESP32-PICO-KIT, ESP32-DevKitC (till v4), ESP32-WROVER-KIT (v4.1 (also known as ESP32-WROVER-KIT-VB), v2, v1 (also known as DevKitJ v1))

You can tell the version of your ESP23-WROVER-KIT version from the module on it: v4.1 are with ESP32-WROVER-B modules, v3 are with ESP32-WROVER modules, while v2 and v1 are with ESP32-WROOM-32 modules.

Refer to *SD Pull-up Requirements* for more technical details of the pullups.

The host initialize the slave into SD mode by first sending CMD0 with DAT3 pin high, or in SPI mode by sending CMD0 with CS pin (the same pin as DAT3) low.

After the initialization, the host can enable the 4-bit SD mode by writing CCCR register 0x07 by CMD52. All the bus detection process are handled by the slave peripheral.

The host has to communicate with the slave by an ESP-slave-specific protocol. The slave driver offers 3 services over Function 1 access by CMD52 and CMD53: (1) a sending FIFO and a receiving FIFO, (2) 52 8-bit R/W registers shared by host and slave, (3) 16 interrupt sources (8 from host to slave, and 8 from slave to host).

Terminology The SDIO slave driver uses the following terms:

- Transfer: a transfer is always started by a command token from the host, and may contain a reply and several data blocks. ESP32 slave software is based on transfers.
- Sending: slave to host transfers.
- Receiving: host to slave transfers.

Note: Register names in *ESP32 Technical Reference Manual* > *SDIO Slave Controller* [PDF] are oriented from the point of view of the host, i.e. 'rx' registers refer to sending, while 'tx' registers refer to receiving. We're not using *tx* or *rx* in the driver to avoid ambiguities.

- FIFO: specific address in Function 1 that can be access by CMD53 to read/write large amount of data. The address is related to the length requested to read from/write to the slave in a single transfer: *requested length* = 0x1F800-address.
- Ownership: When the driver takes ownership of a buffer, it means the driver can randomly read/write the buffer (usually via DMA). The application should not read/write the buffer until the ownership is returned to the application. If the application reads from a buffer owned by a receiving driver, the data read can be random; if the application writes to a buffer owned by a sending driver, the data sent may be corrupted.
- Requested length: The length requested in one transfer determined by the FIFO address.
- Transfer length: The length requested in one transfer determined by the CMD53 byte/block count field.

Note: Requested length is different from the transfer length. ESP32 slave DMA base on the *requested length* rather than the *transfer length*. The *transfer length* should be no shorter than the *requested length*, and the rest part will be filled with 0 (sending) or discard (receiving).

- Receiving buffer size: The buffer size is pre-defined between the host and the slave before communication starts. Slave application has to set the buffer size during initialization by the `recv_buffer_size` member of `sdio_slave_config_t`.
- Interrupts: the esp32 slave support interrupts in two directions: from host to slave (called slave interrupts below) and from slave to host (called host interrupts below). See more in *Interrupts*.
- Registers: specific address in Function 1 access by CMD52 or CMD53.

Communication with ESP SDIO Slave The host should initialize the ESP32 SDIO slave according to the standard SDIO initialization process (Sector 3.1.2 of SDIO Simplified Specification), which is described briefly in *ESP SDIO Slave Initialization*.

Furthermore, there's an ESP32-specific upper-level communication protocol upon the CMD52/CMD53 to Func 1. Please refer to *ESP SDIO Slave Protocol*. There is also a component *ESP Serial Slave Link* for ESP32 master to communicate with ESP32 SDIO slave, see example peripherals/sdio when programming your host.

Interrupts There are interrupts from host to slave, and from slave to host to help communicating conveniently.

Slave Interrupts The host can interrupt the slave by writing any one bit in the register 0x08D. Once any bit of the register is set, an interrupt is raised and the SDIO slave driver calls the callback function defined in the `slave_intr_cb` member in the `sdio_slave_config_t` structure.

Note: The callback function is called in the ISR, do not use any delay, loop or spinlock in the callback.

There's another set of functions can be used. You can call `sdio_slave_wait_int` to wait for an interrupt within a certain time, or call `sdio_slave_clear_int` to clear interrupts from host. The callback function can work with the wait functions perfectly.

Host Interrupts The slave can interrupt the host by an interrupt line (at certain time) which is level sensitive. When the host see the interrupt line pulled down, it may read the slave interrupt status register, to see the interrupt source. Host can clear interrupt bits, or choose to disable a interrupt source. The interrupt line will hold active until all the sources are cleared or disabled.

There are several dedicated interrupt sources as well as general purpose sources. see `sdio_slave_hostint_t` for more information.

Shared Registers There are 52 8-bit R/W shared registers to share information between host and slave. The slave can write or read the registers at any time by `sdio_slave_read_reg` and `sdio_slave_write_reg`. The host can access (R/W) the register by CMD52 or CMD53.

Receiving FIFO When the host is going to send the slave some packets, it has to check whether the slave is ready to receive by reading the buffer number of slave.

To allow the host sending data to the slave, the application has to load buffers to the slave driver by the following steps:

1. Register the buffer by calling `sdio_slave_recv_register_buf`, and get the handle of the registered buffer. The driver will allocate memory for the linked-list descriptor needed to link the buffer onto the hardware. The size of these buffers should equal to the Receiving buffer size.
2. Load buffers onto the driver by passing the buffer handle to `sdio_slave_recv_load_buf`.
3. Get the received data by calling `sdio_slave_recv` or `sdio_slave_recv_packet`. If non-blocking call is needed, set `wait=0`.
   The difference between two APIs is that, `sdio_slave_recv_packet` gives more information about packet, which can consist of several buffers. When `ESP_ERR_NOT_FINISHED` is returned by this API, you should call this API iteratively until the return value is `ESP_OK`. All the continuous buffers returned with `ESP_ERR_NOT_FINISHED`, together with the last buffer returned with `ESP_OK`, belong to one packet from the host. Call `sdio_slave_recv_get_buf` to get the address of the received data, and the actual length received in each buffer. The packet length is the sum of received length of all the buffers in the packet.
   If the host never send data longer than the Receiving buffer size, or you don't care about the packet boundary (e.g. the data is only a byte stream), you can call the simpler version `sdio_slave_recv` instead.
4. Pass the handle of processed buffer back to the driver by `sdio_recv_load_buf` again.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside, the application is responsible to offer new buffers in time. The DMA will automatically store received data to the buffer.

Sending FIFO Each time the slave has data to send, it raises an interrupt and the host will request for the packet length. There are two sending modes:

- Stream Mode: when a buffer is loaded to the driver, the buffer length will be counted into the packet length requested by host in the incoming communications. Regardless previous packets are sent or not. This means the host can get data of several buffers in one transfer.
- Packet Mode: the packet length is updated packet by packet, and only when previous packet is sent. This means that the host can only get data of one buffer in one transfer.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside. Namely, the DMA takes data directly from the buffer provided by the application. The application should not touch the buffer until the sending is finished.

The sending mode can be set in the `sending_mode` member of `sdio_slave_config_t`, and the buffer numbers can be set in the `send_queue_size`. All the buffers are restricted to be no larger than 4092 bytes. Though in the stream mode several buffers can be sent in one transfer, each buffer is still counted as one in the queue.

The application can call `sdio_slave_transmit` to send packets. In this case the function returns when the transfer is successfully done, so the queue is not fully used. When higher effeciency is required, the application can use the following functions instead:

1. Pass buffer information (address, length, as well as an `arg` indicating the buffer) to `sdio_slave_send_queue`. If non-blocking call is needed, set `wait=0`. If the `wait` is not `portMAX_DELAY` (wait until success), application has to check the result to know whether the data is put in to the queue or discard.
2. Call `sdio_slave_send_get_finished` to get and deal with a finished transfer. A buffer should be keep unmodified until returned from `sdio_slave_send_get_finished`. This means the buffer is actually sent to the host, rather than just staying in the queue.

There are several ways to use the `arg` in the queue parameter:

1. Directly point `arg` to a dynamic-allocated buffer, and use the `arg` to free it when transfer finished.
2. Wrap transfer informations in a transfer structure, and point `arg` to the structure. You can use the structure to do more things like:

```
typedef struct {
    uint8_t* buffer;
    size_t   size;
    int      id;
}sdio_transfer_t;

//and send as:
sdio_transfer_t trans = {
    .buffer = ADDRESS_TO_SEND,
    .size = 8,
    .id = 3,  //the 3rd transfer so far
};
sdio_slave_send_queue(trans.buffer, trans.size, &trans, portMAX_DELAY);

//... maybe more transfers are sent here

//and deal with finished transfer as:
sdio_transfer_t* arg = NULL;
sdio_slave_send_get_finished((void**)&arg, portMAX_DELAY);
ESP_LOGI("tag", "(%d) successfully send %d bytes of %p", arg->id, arg->size,␣
↪arg->buffer);
some_post_callback(arg); //do more things
```

3. Working with the receiving part of this driver, point `arg` to the receive buffer handle of this buffer. So that we can directly use the buffer to receive data when it's sent:

```
uint8_t buffer[256]={1,2,3,4,5,6,7,8};
sdio_slave_buf_handle_t handle = sdio_slave_recv_register_buf(buffer);
sdio_slave_send_queue(buffer, 8, handle, portMAX_DELAY);

//... maybe more transfers are sent here

//and load finished buffer to receive as
sdio_slave_buf_handle_t handle = NULL;
sdio_slave_send_get_finished((void**)&handle, portMAX_DELAY);
sdio_slave_recv_load_buf(handle);
```

More about this, see peripherals/sdio.

Application Example

Slave/master communication: peripherals/sdio.

API Reference

Header File

- components/hal/include/hal/sdio_slave_types.h

Enumerations

enum **sdio_slave_hostint_t**

Mask of interrupts sending to the host.

Values:

enumerator **SDIO_SLAVE_HOSTINT_BIT0**

General purpose interrupt bit 0.

enumerator **SDIO_SLAVE_HOSTINT_BIT1**

enumerator **SDIO_SLAVE_HOSTINT_BIT2**

enumerator **SDIO_SLAVE_HOSTINT_BIT3**

enumerator **SDIO_SLAVE_HOSTINT_BIT4**

enumerator **SDIO_SLAVE_HOSTINT_BIT5**

enumerator **SDIO_SLAVE_HOSTINT_BIT6**

enumerator **SDIO_SLAVE_HOSTINT_BIT7**

enumerator **SDIO_SLAVE_HOSTINT_SEND_NEW_PACKET**

New packet available.

enum **sdio_slave_timing_t**

Timing of SDIO slave.

Values:

enumerator **SDIO_SLAVE_TIMING_PSEND_PSAMPLE**

Send at posedge, and sample at posedge. Default value for HS mode. If :c:macro:`SDIO_SLAVE_FLAG_HIGH_SPEED` is specified in :cpp:class:*sdio_slave_config_t*, this should be selected. Normally there' s no problem using this to work in DS mode.

enumerator **SDIO_SLAVE_TIMING_NSEND_PSAMPLE**

Send at negedge, and sample at posedge. Default value for DS mode and below. If :c:macro:`SDIO_SLAVE_FLAG_DEFAULT_SPEED` is specified in :cpp:class:*sdio_slave_config_t*, this should be selected.

enumerator **SDIO_SLAVE_TIMING_PSEND_NSAMPLE**

Send at posedge, and sample at negedge.

enumerator **SDIO_SLAVE_TIMING_NSEND_NSAMPLE**

Send at negedge, and sample at negedge.

enum **sdio_slave_sending_mode_t**

Configuration of SDIO slave mode.

Values:

enumerator **SDIO_SLAVE_SEND_STREAM**

Stream mode, all packets to send will be combined as one if possible.

enumerator **SDIO_SLAVE_SEND_PACKET**

Packet mode, one packets will be sent one after another (only increase packet_len if last packet sent).

Header File

- components/driver/include/driver/sdio_slave.h

Functions

esp_err_t **sdio_slave_initialize**(*sdio_slave_config_t* *config)

Initialize the sdio slave driver

Parameters **config** – Configuration of the sdio slave driver.

Returns

- ESP_ERR_NOT_FOUND if no free interrupt found.
- ESP_ERR_INVALID_STATE if already initialized.
- ESP_ERR_NO_MEM if fail due to memory allocation failed.
- ESP_OK if success

void **sdio_slave_deinit**(void)

De-initialize the sdio slave driver to release the resources.

esp_err_t **sdio_slave_start**(void)

Start hardware for sending and receiving, as well as set the IOREADY1 to 1.

Note: The driver will continue sending from previous data and PKT_LEN counting, keep data received as well as start receiving from current TOKEN1 counting. See `sdio_slave_reset`.

Returns

- ESP_ERR_INVALID_STATE if already started.

- ESP_OK otherwise.

void **sdio_slave_stop**(void)

Stop hardware from sending and receiving, also set IOREADY1 to 0.

Note: this will not clear the data already in the driver, and also not reset the PKT_LEN and TOKEN1 counting. Call `sdio_slave_reset` to do that.

---

esp_err_t **sdio_slave_reset**(void)

Clear the data still in the driver, as well as reset the PKT_LEN and TOKEN1 counting.

Returns always return ESP_OK.

sdio_slave_buf_handle_t **sdio_slave_recv_register_buf**(uint8_t *start)

Register buffer used for receiving. All buffers should be registered before used, and then can be used (again) in the driver by the handle returned.

Note: The driver will use and only use the amount of space specified in the `recv_buffer_size` member set in the *`sdio_slave_config_t`*. All buffers should be larger than that. The buffer is used by the DMA, so it should be DMA capable and 32-bit aligned.

Parameters **start** –The start address of the buffer.
Returns The buffer handle if success, otherwise NULL.

esp_err_t **sdio_slave_recv_unregister_buf**(*sdio_slave_buf_handle_t* handle)

Unregister buffer from driver, and free the space used by the descriptor pointing to the buffer.

Parameters **handle** –Handle to the buffer to release.
Returns ESP_OK if success, ESP_ERR_INVALID_ARG if the handle is NULL or the buffer is being used.

esp_err_t **sdio_slave_recv_load_buf**(*sdio_slave_buf_handle_t* handle)

Load buffer to the queue waiting to receive data. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Parameters **handle** –Handle to the buffer ready to receive data.
Returns

- ESP_ERR_INVALID_ARG if invalid handle or the buffer is already in the queue. Only after the buffer is returened by `sdio_slave_recv` can you load it again.
- ESP_OK if success

esp_err_t **sdio_slave_recv_packet**(*sdio_slave_buf_handle_t* *handle_ret, TickType_t wait)

Get buffer of received data if exist with packet information. The driver returns the ownership of the buffer to the app.

When you see return value is `ESP_ERR_NOT_FINISHED`, you should call this API iteratively until the return value is `ESP_OK`. All the continuous buffers returned with `ESP_ERR_NOT_FINISHED`, together with the last buffer returned with `ESP_OK`, belong to one packet from the host.

You can call simpler `sdio_slave_recv` instead, if the host never send data longer than the Receiving buffer size, or you don't care about the packet boundary (e.g. the data is only a byte stream).

Note: Call `sdio_slave_load_buf` with the handle to re-load the buffer onto the link list, and receive with the same buffer again. The address and length of the buffer got here is the same as got from `sdio_slave_get_buffer`.

Parameters

- **handle_ret** –Handle of the buffer holding received data. Use this handle in `sdio_slave_recv_load_buf()` to receive in the same buffer again.
- **wait** –Time to wait before data received.

Returns

- ESP_ERR_INVALID_ARG if handle_ret is NULL
- ESP_ERR_TIMEOUT if timeout before receiving new data
- ESP_ERR_NOT_FINISHED if returned buffer is not the end of a packet from the host, should call this API again until the end of a packet
- ESP_OK if success

esp_err_t **sdio_slave_recv**(*sdio_slave_buf_handle_t* *handle_ret, uint8_t **out_addr, size_t *out_len, TickType_t wait)

Get received data if exist. The driver returns the ownership of the buffer to the app.

Note: Call `sdio_slave_load_buf` with the handle to re-load the buffer onto the link list, and receive with the same buffer again. The address and length of the buffer got here is the same as got from `sdio_slave_get_buffer`.

Parameters

- **handle_ret** –Handle to the buffer holding received data. Use this handle in `sdio_slave_recv_load_buf` to receive in the same buffer again.
- **out_addr** –**[out]** Output of the start address, set to NULL if not needed.
- **out_len** –**[out]** Actual length of the data in the buffer, set to NULL if not needed.
- **wait** –Time to wait before data received.

Returns

- ESP_ERR_INVALID_ARG if handle_ret is NULL
- ESP_ERR_TIMEOUT if timeout before receiving new data
- ESP_OK if success

uint8_t ***sdio_slave_recv_get_buf**(*sdio_slave_buf_handle_t* handle, size_t *len_o)

Retrieve the buffer corresponding to a handle.

Parameters

- **handle** –Handle to get the buffer.
- **len_o** –Output of buffer length

Returns buffer address if success, otherwise NULL.

esp_err_t **sdio_slave_send_queue**(uint8_t *addr, size_t len, void *arg, TickType_t wait)

Put a new sending transfer into the send queue. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Parameters

- **addr** –Address for data to be sent. The buffer should be DMA capable and 32-bit aligned.
- **len** –Length of the data, should not be longer than 4092 bytes (may support longer in the future).
- **arg** –Argument to returned in `sdio_slave_send_get_finished`. The argument can be used to indicate which transaction is done, or as a parameter for a callback. Set to NULL if not needed.
- **wait** –Time to wait if the buffer is full.

Returns

- ESP_ERR_INVALID_ARG if the length is not greater than 0.
- ESP_ERR_TIMEOUT if the queue is still full until timeout.
- ESP_OK if success.

esp_err_t **sdio_slave_send_get_finished**(void **out_arg, TickType_t wait)

Return the ownership of a finished transaction.

Parameters

- **out_arg** –Argument of the finished transaction. Set to NULL if unused.
- **wait** –Time to wait if there’ s no finished sending transaction.

Returns ESP_ERR_TIMEOUT if no transaction finished, or ESP_OK if succeed.

esp_err_t **sdio_slave_transmit**(uint8_t *addr, size_t len)

Start a new sending transfer, and wait for it (blocked) to be finished.

Parameters
- **addr** –Start address of the buffer to send
- **len** –Length of buffer to send.

Returns
- ESP_ERR_INVALID_ARG if the length of descriptor is not greater than 0.
- ESP_ERR_TIMEOUT if the queue is full or host do not start a transfer before timeout.
- ESP_OK if success.

uint8_t **sdio_slave_read_reg**(int pos)

Read the spi slave register shared with host.

Note: register 28 to 31 are reserved for interrupt vector.

Parameters **pos** –register address, 0-27 or 32-63.
Returns value of the register.

esp_err_t **sdio_slave_write_reg**(int pos, uint8_t reg)

Write the spi slave register shared with host.

Note: register 29 and 31 are used for interrupt vector.

Parameters
- **pos** –register address, 0-11, 14-15, 18-19, 24-27 and 32-63, other address are reserved.
- **reg** –the value to write.

Returns ESP_ERR_INVALID_ARG if address wrong, otherwise ESP_OK.

sdio_slave_hostint_t **sdio_slave_get_host_intena**(void)

Get the interrupt enable for host.

Returns the interrupt mask.

void **sdio_slave_set_host_intena**(*sdio_slave_hostint_t* mask)

Set the interrupt enable for host.

Parameters **mask** –Enable mask for host interrupt.

esp_err_t **sdio_slave_send_host_int**(uint8_t pos)

Interrupt the host by general purpose interrupt.

Parameters **pos** –Interrupt num, 0-7.
Returns
- ESP_ERR_INVALID_ARG if interrupt num error
- ESP_OK otherwise

void **sdio_slave_clear_host_int**(*sdio_slave_hostint_t* mask)

Clear general purpose interrupt to host.

Parameters **mask** –Interrupt bits to clear, by bit mask.

esp_err_t **sdio_slave_wait_int** (int pos, TickType_t wait)

Wait for general purpose interrupt from host.

Note: this clears the interrupt at the same time.

Parameters

- **pos** –Interrupt source number to wait for. is set.
- **wait** –Time to wait before interrupt triggered.

Returns ESP_OK if success, ESP_ERR_TIMEOUT if timeout.

Structures

struct **sdio_slave_config_t**

Configuration of SDIO slave.

Public Members

sdio_slave_timing_t **timing**

timing of sdio_slave. see `sdio_slave_timing_t`.

sdio_slave_sending_mode_t **sending_mode**

mode of sdio_slave. `SDIO_SLAVE_MODE_STREAM` if the data needs to be sent as much as possible; `SDIO_SLAVE_MODE_PACKET` if the data should be sent in packets.

int **send_queue_size**

max buffers that can be queued before sending.

size_t **recv_buffer_size**

If buffer_size is too small, it costs more CPU time to handle larger number of buffers. If buffer_size is too large, the space larger than the transaction length is left blank but still counts a buffer, and the buffers are easily run out. Should be set according to length of data really transferred. All data that do not fully fill a buffer is still counted as one buffer. E.g. 10 bytes data costs 2 buffers if the size is 8 bytes per buffer. Buffer size of the slave pre-defined between host and slave before communication. All receive buffer given to the driver should be larger than this.

sdio_event_cb_t **event_cb**

when the host interrupts slave, this callback will be called with interrupt number (0-7).

uint32_t **flags**

Features to be enabled for the slave, combinations of `SDIO_SLAVE_FLAG_*`.

Macros

SDIO_SLAVE_RECV_MAX_BUFFER

SDIO_SLAVE_FLAG_DAT2_DISABLED

It is required by the SD specification that all 4 data lines should be used and pulled up even in 1-bit mode or SPI mode. However, as a feature, the user can specify this flag to make use of DAT2 pin in 1-bit mode. Note that the host cannot read CCCR registers to know we don't support 4-bit mode anymore, please do this at your own risk.

SDIO_SLAVE_FLAG_HOST_INTR_DISABLED

The DAT1 line is used as the interrupt line in SDIO protocol. However, as a feature, the user can specify this flag to make use of DAT1 pin of the slave in 1-bit mode. Note that the host has to do polling to the interrupt registers to know whether there are interrupts from the slave. And it cannot read CCCR registers to know we don't support 4-bit mode anymore, please do this at your own risk.

SDIO_SLAVE_FLAG_INTERNAL_PULLUP

Enable internal pullups for enabled pins. It is required by the SD specification that all the 4 data lines should be pulled up even in 1-bit mode or SPI mode. Note that the internal pull-ups are not sufficient for stable communication, please do connect external pull-ups on the bus. This is only for example and debug use.

SDIO_SLAVE_FLAG_DEFAULT_SPEED

Disable the highspeed support of the hardware.

SDIO_SLAVE_FLAG_HIGH_SPEED

Enable the highspeed support of the hardware. This is the default option. The host will see highspeed capability, but the mode actually used is determined by the host.

Type Definitions

typedef void (***sdio_event_cb_t**)(uint8_t event)

typedef void ***sdio_slave_buf_handle_t**

Handle of a receive buffer, register a handle by calling `sdio_slave_recv_register_buf`. Use the handle to load the buffer to the driver, or call `sdio_slave_recv_unregister_buf` if it is no longer used.

2.6.19 Sigma-Delta Modulation (SDM)

Introduction

ESP32 has a second-order sigma-delta modulator, which can generate independent PDM pulses to multiple channels. Please refer to the TRM to check how many hardware channels are available.[1]

Typically, a Sigma-Delta modulated channel can be used in scenarios like:

- LED dimming
- Simple DAC (8-bit), with the help of an active RC low-pass filter
- Class D amplifier, with the help of a half-bridge or full-bridge circuit plus an LC low-pass filter

Functional Overview

The following sections of this document cover the typical steps to install and operate a SDM channel:

- *Resource Allocation* - covers which parameters should be set up to get a channel handle and how to recycle the resources when it finishes working.
- *Enable and Disable Channel* - covers how to enable and disable the channel.
- *Set Equivalent Duty Cycle* - describes how to set the equivalent duty cycle of the PDM pulses.
- *Power Management* - describes how different source clock selections can affect power consumption.
- *IRAM Safe* - lists which functions are supposed to work even when the cache is disabled.
- *Thread Safety* - lists which APIs are guaranteed to be thread safe by the driver.

[1] Different ESP chip series might have different numbers of SDM channels. Please refer to Chapter GPIO and IOMUX in ESP32 Technical Reference Manual for more details. The driver won't forbid you from applying for more channels, but it will return error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g. `sdm_new_channel()`).

- *Kconfig Options* - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

Resource Allocation A SDM channel is represented by `sdm_channel_handle_t`. Each channel is capable to output the binary, hardware generated signal with the sigma-delta modulation. The driver manages all available channels in a pool, so that users don't need to manually assign a fixed channel to a GPIO.

To install a SDM channel, you should call `sdm_new_channel()` to get a channel handle. Channel specific configurations are passed in the `sdm_config_t` structure:

- `sdm_config_t::gpio_num` sets the GPIO that the PDM pulses will output from
- `sdm_config_t::clk_src` selects the source clock for the SDM module. Note that, all channels should select the same clock source.
- `sdm_config_t::sample_rate_hz` sets the sample rate of the SDM module.
- `sdm_config_t::invert_out` sets whether to invert the output signal.
- `sdm_config_t::io_loop_back` is for debugging purposes only. It enables both the GPIO's input and output ability through the GPIO matrix peripheral.

The function `sdm_new_channel()` can fail due to various errors such as insufficient memory, invalid arguments, etc. Specifically, when there are no more free channels (i.e. all hardware SDM channels have been used up), then `ESP_ERR_NOT_FOUND` will be returned.

If a previously created SDM channel is no longer required, you should recycle it by calling `sdm_del_channel()`. It allows the underlying HW channel to be used for other purposes. Before deleting a SDM channel handle, you should disable it by `sdm_channel_disable()` in advance or make sure it has not enabled yet by `sdm_channel_enable()`.

Creating a SDM Channel with Sample Rate of 1MHz

```
sdm_channel_handle_t chan = NULL;
sdm_config_t config = {
    .clk_src = SDM_CLK_SRC_DEFAULT,
    .sample_rate_hz = 1 * 1000 * 1000,
    .gpio_num = 0,
};
ESP_ERROR_CHECK(sdm_new_channel(&config, &chan));
```

Enable and Disable Channel Before doing further IO control to the SDM channel, you should enable it first, by calling `sdm_channel_enable()`. Internally, this function will:

- switch the channel state from **init** to **enable**
- acquire a proper power management lock is a specific clock source (e.g. APB clock) is selected. See also *Power management* for more information.

On the contrary, calling `sdm_channel_disable()` will do the opposite, that is, put the channel back to the **init** state and release the power management lock.

Set Equivalent Duty Cycle For the output PDM signals, the duty cycle refers to the percentage of high level cycles to the whole statistical period. The average output voltage from the channel is calculated by `Vout = VDD_IO / 256 * duty + VDD_IO / 2`. Thus the range of the `duty` input parameter of `sdm_channel_set_duty()` is from -128 to 127 (eight bit signed integer). For example,if zero value is set, then the output signal's duty will be about 50%.

Power Management When power management is enabled (i.e. *CONFIG_PM_ENABLE* is on), the system will adjust the APB frequency before going into light sleep, thus potentially changing the sample rate of the sigma-delta modulator.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type *ESP_PM_APB_FREQ_MAX*. Whenever the driver creates a SDM channel instance that has selected *SDM_CLK_SRC_APB* as its clock source, the driver will guarantee that the power management lock is acquired when enable the channel by *sdm_channel_enable()*. Likewise, the driver releases the lock when *sdm_channel_disable()* is called for that channel.

IRAM Safe There's a Kconfig option *CONFIG_SDM_CTRL_FUNC_IN_IRAM* that can put commonly used IO control functions into IRAM as well. So that these functions can also be executable when the cache is disabled. These IO control functions are listed as follows:

- *sdm_channel_set_duty()*

Thread Safety The factory function *sdm_new_channel()* is guaranteed to be thread safe by the driver, which means, user can call it from different RTOS tasks without protection by extra locks. The following functions are allowed to run under ISR context, the driver uses a critical section to prevent them being called concurrently in both task and ISR.

- *sdm_channel_set_duty()*

Other functions that take the *sdm_channel_handle_t* as the first positional parameter, are not treated as thread safe. Which means the user should avoid calling them from multiple tasks.

Kconfig Options

- *CONFIG_SDM_CTRL_FUNC_IN_IRAM* controls where to place the SDM channel control functions (IRAM or Flash), see *IRAM Safe* for more information.
- *CONFIG_SDM_ENABLE_DEBUG_LOG* is used to enabled the debug log output. Enable this option will increase the firmware binary size.

Convert to analog signal (Optional)

Typically, if the sigma-delta signal is connected to an LED, you don't have to add any filter between them (because our eyes are a low pass filter naturally). However, if you want to check the real voltage or watch the analog waveform, you need to design an analog low pass filter. Also, it is recommended to use an active filter instead of a passive filter to gain better isolation and not lose too much voltage.

For example, you can take the following Sallen-Key topology Low Pass Filter as a reference.

Application Example

- LED driven by a GPIO that is modulated with Sigma-Delta: peripherals/sigma_delta.

API Reference

Header File

- components/driver/include/driver/sdm.h

Functions

esp_err_t **sdm_new_channel**(const *sdm_config_t* *config, *sdm_channel_handle_t* *ret_chan)

Create a new Sigma Delta channel.

Parameters

- **config** –**[in]** SDM configuration
- **ret_chan** –**[out]** Returned SDM channel handle

Returns

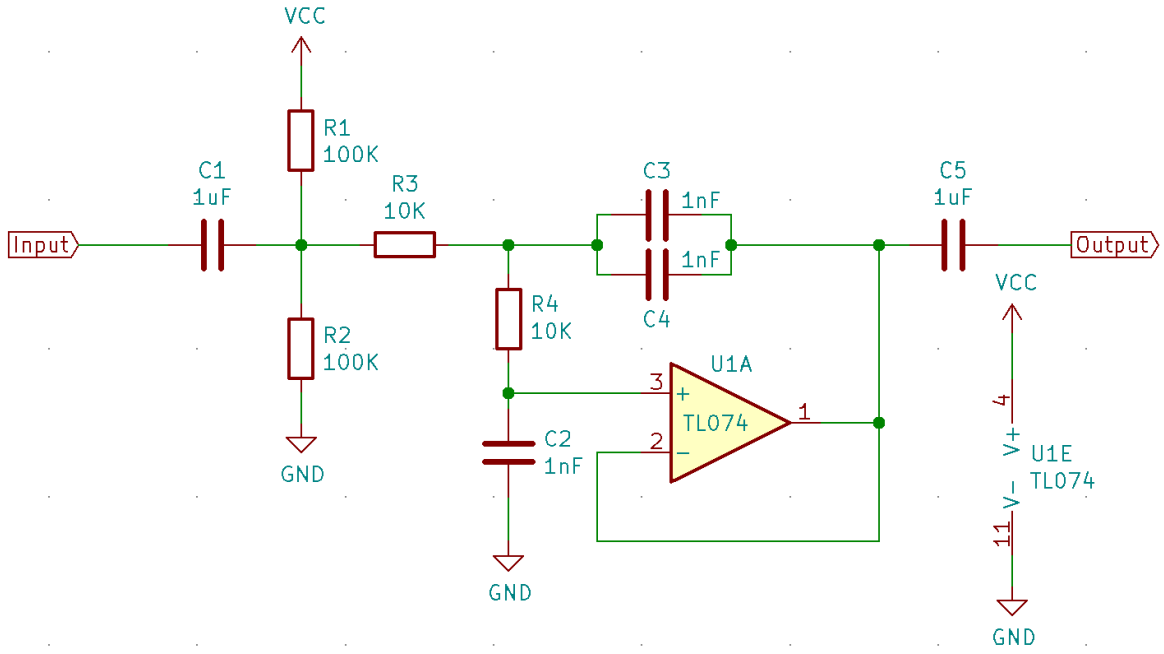


Fig. 21: Sallen-Key Low Pass Filter

- ESP_OK: Create SDM channel successfully
- ESP_ERR_INVALID_ARG: Create SDM channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create SDM channel failed because out of memory
- ESP_ERR_NOT_FOUND: Create SDM channel failed because all channels are used up and no more free one
- ESP_FAIL: Create SDM channel failed because of other error

esp_err_t **sdm_del_channel** (*sdm_channel_handle_t* chan)

Delete the Sigma Delta channel.

Parameters **chan** –**[in]** SDM channel created by `sdm_new_channel`

Returns

- ESP_OK: Delete the SDM channel successfully
- ESP_ERR_INVALID_ARG: Delete the SDM channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete the SDM channel failed because the channel is not in init state
- ESP_FAIL: Delete the SDM channel failed because of other error

esp_err_t **sdm_channel_enable** (*sdm_channel_handle_t* chan)

Enable the Sigma Delta channel.

Note: This function will transit the channel state from init to enable.

Note: This function will acquire a PM lock, if a specific source clock (e.g. APB) is selected in the *`sdm_config_t`*, while `CONFIG_PM_ENABLE` is enabled.

Parameters **chan** –**[in]** SDM channel created by `sdm_new_channel`

Returns

- ESP_OK: Enable SDM channel successfully
- ESP_ERR_INVALID_ARG: Enable SDM channel failed because of invalid argument

- ESP_ERR_INVALID_STATE: Enable SDM channel failed because the channel is already enabled
- ESP_FAIL: Enable SDM channel failed because of other error

esp_err_t **sdm_channel_disable** (*sdm_channel_handle_t* chan)

Disable the Sigma Delta channel.

Note: This function will do the opposite work to the `sdm_channel_enable()`

Parameters **chan** –**[in]** SDM channel created by `sdm_new_channel`

Returns

- ESP_OK: Disable SDM channel successfully
- ESP_ERR_INVALID_ARG: Disable SDM channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable SDM channel failed because the channel is not enabled yet
- ESP_FAIL: Disable SDM channel failed because of other error

esp_err_t **sdm_channel_set_duty** (*sdm_channel_handle_t* chan, int8_t duty)

Set the duty cycle of the PDM output signal.

Note: For PDM signals, duty cycle refers to the percentage of high level cycles to the whole statistical period. The average output voltage could be Vout = VDD_IO / 256 * duty + VDD_IO / 2

Note: If the duty is set to zero, the output signal is like a 50% duty cycle square wave, with a frequency around (sample_rate_hz / 4).

Note: The duty is proportional to the equivalent output voltage after a low-pass-filter.

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if `CONFIG_SDM_CTRL_FUNC_IN_IRAM` is on, so that it's allowed to be executed when Cache is disabled

---

Parameters

- **chan** –**[in]** SDM channel created by `sdm_new_channel`
- **duty** –**[in]** Equivalent duty cycle of the PDM output signal, ranges from -128 to 127. But the range of [-90, 90] can provide a better randomness.

Returns

- ESP_OK: Set duty cycle successfully
- ESP_ERR_INVALID_ARG: Set duty cycle failed because of invalid argument
- ESP_FAIL: Set duty cycle failed because of other error

Structures

struct **sdm_config_t**

Sigma Delta channel configuration.

Public Members

int **gpio_num**

GPIO number

sdm_clock_source_t **clk_src**

Clock source

uint32_t **sample_rate_hz**

Sample rate in Hz, it determines how frequent the modulator outputs a pulse

uint32_t **invert_out**

Whether to invert the output signal

uint32_t **io_loop_back**

For debug/test, the signal output from the GPIO will be fed to the input path as well

struct *sdm_config_t*::[anonymous] **flags**

Extra flags

Type Definitions

typedef struct sdm_channel_t ***sdm_channel_handle_t**

Type of Sigma Delta channel handle.

Header File

- components/hal/include/hal/sdm_types.h

Type Definitions

typedef *soc_periph_sdm_clk_src_t* **sdm_clock_source_t**

2.6.20 SPI Master Driver

SPI Master driver is a program that controls ESP32's SPI peripherals while they function as masters.

Overview of ESP32's SPI peripherals

ESP32 integrates 4 SPI peripherals.

- SPI0 and SPI1 are used internally to access the ESP32's attached flash memory. Both controllers share the same SPI bus signals, and there is an arbiter to determine which can access the bus.
  There are quite a few limitations when using SPI Master driver on the SPI1 bus, see *Notes on Using the SPI Master driver on SPI1 Bus*.
- SPI2 and SPI3 are general purpose SPI controllers, sometimes referred to as HSPI and VSPI, respectively. They are open to users. SPI2 and SPI3 have independent bus signals with the same respective names. Each bus has three CS lines to drive up to same number of SPI slaves.

Terminology

The terms used in relation to the SPI master driver are given in the table below.

Term	Definition
Host	The SPI controller peripheral inside ESP32 that initiates SPI transmissions over the bus, and acts as an SPI Master.
Device	SPI slave device. An SPI bus may be connected to one or more Devices. Each Device shares the MOSI, MISO and SCLK signals but is only active on the bus when the Host asserts the Device's individual CS line.
Bus	A signal bus, common to all Devices connected to one Host. In general, a bus includes the following lines: MISO, MOSI, SCLK, one or more CS lines, and, optionally, QUADWP and QUADHD. So Devices are connected to the same lines, with the exception that each Device has its own CS line. Several Devices can also share one CS line if connected in the daisy-chain manner.
MOSI	Master Out, Slave In, a.k.a. D. Data transmission from a Host to Device. Also data0 signal in Octal/OPI mode.
MISO	Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host. Also data1 signal in Octal/OPI mode.
SCLK	Serial Clock. Oscillating signal generated by a Host that keeps the transmission of data bits in sync.
CS	Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data.
QUADWP	Write Protect signal. Used for 4-bit (qio/qout) transactions. Also for data2 signal in Octal/OPI mode.
QUADHD	Hold signal. Used for 4-bit (qio/qout) transactions. Also for data3 signal in Octal/OPI mode.
DATA4	Data4 signal in Octal/OPI mode.
DATA5	Data5 signal in Octal/OPI mode.
DATA6	Data6 signal in Octal/OPI mode.
DATA7	Data7 signal in Octal/OPI mode.
Assertion	The action of activating a line.
De-assertion	The action of returning the line back to inactive (back to idle) status.
Transaction	One instance of a Host asserting a CS line, transferring data to and from a Device, and de-asserting the CS line. Transactions are atomic, which means they can never be interrupted by another transaction.
Launch edge	Edge of the clock at which the source register *launches* the signal onto the line.
Latch edge	Edge of the clock at which the destination register *latches in* the signal.

Driver Features

The SPI master driver governs communications of Hosts with Devices. The driver supports the following features:

- Multi-threaded environments
- Transparent handling of DMA transfers while reading and writing data
- Automatic time-division multiplexing of data coming from different Devices on the same signal bus, see *SPI Bus Lock*.

Warning: The SPI master driver has the concept of multiple Devices connected to a single bus (sharing a single ESP32 SPI peripheral). As long as each Device is accessed by only one task, the driver is thread safe. However, if multiple tasks try to access the same SPI Device, the driver is **not thread-safe**. In this case, it is recommended to either:

- Refactor your application so that each SPI peripheral is only accessed by a single task at a time.
- Add a mutex lock around the shared Device using `xSemaphoreCreateMutex`.

SPI Features

SPI Master

SPI Bus Lock To realize the multiplexing of different devices from different drivers, including SPI Master, SPI Flash, etc., an SPI bus lock is applied on each SPI bus. Drivers can attach their devices onto the bus with the arbitration of the lock.

Each bus lock is initialized with a BG (background) service registered. All devices that request transactions on the bus should wait until the BG is successfully disabled.

- For SPI1 bus, the BG is the cache. The bus lock will disable the cache before device operations start, and enable it again after device releases the lock. No devices on SPI1 is allowed to use ISR, since it is meaningless for the task to yield to other tasks when the cache is disabled.
  There are quite a few limitations when using SPI Master driver on the SPI1 bus. See *Notes on Using the SPI Master driver on SPI1 Bus*.
- For other buses, the driver may register its ISR as the BG. When a device task requests for exclusive use of the bus, the bus lock will block the task and try to disable ISR. After ISR is successfully disabled, the bus lock will then unblock the device task and allow it to exclusively use the bus. When the task releases the lock, the lock will also try to resume ISR if there are pending transactions in ISR.

SPI Transactions

An SPI bus transaction consists of five phases which can be found in the table below. Any of these phases can be skipped.

Phase	Description
Command	In this phase, a command (0-16 bit) is written to the bus by the Host.
Address	In this phase, an address (0-64 bit) is transmitted over the bus by the Host.
Write	Host sends data to a Device. This data follows the optional command and address phases and is indistinguishable from them at the electrical level.
Dummy	This phase is configurable and is used to meet the timing requirements.
Read	Device sends data to its Host.

The attributes of a transaction are determined by the bus configuration structure `spi_bus_config_t`, device configuration structure `spi_device_interface_config_t`, and transaction configuration structure `spi_transaction_t`.

An SPI Host can send full-duplex transactions, during which the read and write phases occur simultaneously. The total transaction length is determined by the sum of the following members:

- `spi_device_interface_config_t::command_bits`
- `spi_device_interface_config_t::address_bits`
- `spi_transaction_t::length`

While the member `spi_transaction_t::rxlength` only determines the length of data received into the buffer.

In half-duplex transactions, the read and write phases are not simultaneous (one direction at a time). The lengths of the write and read phases are determined by `spi_transaction_t::length` and `spi_transaction_t::rxlength` respectively.

The command and address phases are optional, as not every SPI device requires a command and/or address. This is reflected in the Device' s configuration: if `spi_device_interface_config_t::command_bits` and/or `spi_device_interface_config_t::address_bits` are set to zero, no command or address phase will occur.

The read and write phases can also be optional, as not every transaction requires both writing and reading data. If *spi_transaction_t::rx_buffer* is NULL and *SPI_TRANS_USE_RXDATA* is not set, the read phase is skipped. If *spi_transaction_t::tx_buffer* is NULL and *SPI_TRANS_USE_TXDATA* is not set, the write phase is skipped.

The driver supports two types of transactions: the interrupt transactions and polling transactions. The programmer can choose to use a different transaction type per Device. If your Device requires both transaction types, see *Notes on Sending Mixed Transactions to the Same Device*.

Interrupt Transactions Interrupt transactions will block the transaction routine until the transaction completes, thus allowing the CPU to run other tasks.

An application task can queue multiple transactions, and the driver will automatically handle them one-by-one in the interrupt service routine (ISR). It allows the task to switch to other procedures until all the transactions complete.

Polling Transactions Polling transactions do not use interrupts. The routine keeps polling the SPI Host' s status bit until the transaction is finished.

All the tasks that use interrupt transactions can be blocked by the queue. At this point, they will need to wait for the ISR to run twice before the transaction is finished. Polling transactions save time otherwise spent on queue handling and context switching, which results in smaller transaction duration. The disadvantage is that the CPU is busy while these transactions are in progress.

The *spi_device_polling_end()* routine needs an overhead of at least 1 us to unblock other tasks when the transaction is finished. It is strongly recommended to wrap a series of polling transactions using the functions *spi_device_acquire_bus()* and *spi_device_release_bus()* to avoid the overhead. For more information, see *Bus Acquiring*.

Transaction Line Mode Supported line modes for ESP32 are listed as follows, to make use of these modes, set the member *flags* in the struct *spi_transaction_t* as shown in the *Transaction Flag* column. If you want to check if corresponding IO pins are set or not, set the member *flags* in the *spi_bus_config_t* as shown in the *Bus IO setting Flag* column.

Mode name	Command Line Width	Address Line Width	Data Line Width	Transaction Flag	Bus IO setting Flag
Normal SPI	1	1	1	0	0
Dual Output	1	1	2	SPI_TRANS_MODE_DIO	SPICOMMON_BUSFLAG_DUAL
Dual I/O	1	2	2	SPI_TRANS_MODE_DIO | SPI_TRANS_MULTILINE_ADDR	
Quad Output	1	1	4	SPI_TRANS_MODE_QIO	SPICOMMON_BUSFLAG_QUAD
Quad I/O	1	4	4	SPI_TRANS_MODE_QIO | SPI_TRANS_MULTILINE_ADDR	

Command and Address Phases During the command and address phases, the members *spi_transaction_t::cmd* and *spi_transaction_t::addr* are sent to the bus, nothing is read at this time. The default lengths of the command and address phases are set in *spi_device_interface_config_t* by calling *spi_bus_add_device()*. If the flags *SPI_TRANS_VARIABLE_CMD* and *SPI_TRANS_VARIABLE_ADDR* in the member *spi_transaction_t::flags* are not set, the driver automatically sets the length of these phases to default values during Device initialization.

If the lengths of the command and address phases need to be variable, declare the struct *spi_transaction_ext_t*, set the flags *SPI_TRANS_VARIABLE_CMD* and/or

SPI_TRANS_VARIABLE_ADDR in the member *spi_transaction_ext_t::base* and configure the rest of base as usual. Then the length of each phase will be equal to *spi_transaction_ext_t::command_bits* and *spi_transaction_ext_t::address_bits* set in the struct *spi_transaction_ext_t*.

If the command and address phase need to be as the same number of lines as data phase, you need to set *SPI_TRANS_MULTILINE_CMD* and/or *SPI_TRANS_MULTILINE_ADDR* to the *flags* member in the struct *spi_transaction_t*. Also see *Transaction Line Mode*.

Write and Read Phases Normally, the data that needs to be transferred to or from a Device will be read from or written to a chunk of memory indicated by the members *spi_transaction_t::rx_buffer* and *spi_transaction_t::tx_buffer*. If DMA is enabled for transfers, the buffers are required to be:

1. Allocated in DMA-capable internal memory. If *external PSRAM is enabled*, this means using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.
2. 32-bit aligned (staring from a 32-bit boundary and having a length of multiples of 4 bytes).

If these requirements are not satisfied, the transaction efficiency will be affected due to the allocation and copying of temporary buffers.

If using more than one data lines to transmit, please set *SPI_DEVICE_HALFDUPLEX* flag for the member *flags* in the struct *spi_device_interface_config_t*. And the member *flags* in the struct *spi_transaction_t* should be set as described in *Transaction Line Mode*.

Note: Half-duplex transactions with both read and write phases are not supported when using DMA. For details and workarounds, see *Known Issues*.

Bus Acquiring Sometimes you might want to send SPI transactions exclusively and continuously so that it takes as little time as possible. For this, you can use bus acquiring, which helps to suspend transactions (both polling or interrupt) to other devices until the bus is released. To acquire and release a bus, use the functions *spi_device_acquire_bus()* and *spi_device_release_bus()*.

Driver Usage

- Initialize an SPI bus by calling the function *spi_bus_initialize()*. Make sure to set the correct I/O pins in the struct *spi_bus_config_t*. Set the signals that are not needed to `-1`.
- Register a Device connected to the bus with the driver by calling the function *spi_bus_add_device()*. Make sure to configure any timing requirements the device might need with the parameter `dev_config`. You should now have obtained the Device's handle which will be used when sending a transaction to it.
- To interact with the Device, fill one or more *spi_transaction_t* structs with any transaction parameters required. Then send the structs either using a polling transaction or an interrupt transaction:
  - ***Interrupt*** Either queue all transactions by calling the function *spi_device_queue_trans()* and, at a later time, query the result using the function *spi_device_get_trans_result()*, or handle all requests synchronously by feeding them into *spi_device_transmit()*.
  - ***Polling*** Call the function *spi_device_polling_transmit()* to send polling transactions. Alternatively, if you want to insert something in between, send the transactions by using *spi_device_polling_start()* and *spi_device_polling_end()*.
- (Optional) To perform back-to-back transactions with a Device, call the function *spi_device_acquire_bus()* before sending transactions and *spi_device_release_bus()* after the transactions have been sent.
- (Optional) To unload the driver for a certain Device, call *spi_bus_remove_device()* with the Device handle as an argument.
- (Optional) To remove the driver for a bus, make sure no more drivers are attached and call *spi_bus_free()*.

The example code for the SPI master driver can be found in the peripherals/spi_master directory of ESP-IDF examples.

Transactions with Data Not Exceeding 32 Bits When the transaction data size is equal to or less than 32 bits, it will be sub-optimal to allocate a buffer for the data. The data can be directly stored in the transaction struct instead. For transmitted data, it can be achieved by using the *spi_transaction_t::tx_data* member and setting the *SPI_TRANS_USE_TXDATA* flag on the transmission. For received data, use *spi_transaction_t::rx_data* and set *SPI_TRANS_USE_RXDATA*. In both cases, do not touch the *spi_transaction_t::tx_buffer* or *spi_transaction_t::rx_buffer* members, because they use the same memory locations as *spi_transaction_t::tx_data* and *spi_transaction_t::rx_data*.

Transactions with Integers Other Than `uint8_t` An SPI Host reads and writes data into memory byte by byte. By default, data is sent with the most significant bit (MSB) first, as LSB first used in rare cases. If a value less than 8 bits needs to be sent, the bits should be written into memory in the MSB first manner.

For example, if `0b00010` needs to be sent, it should be written into a `uint8_t` variable, and the length for reading should be set to 5 bits. The Device will still receive 8 bits with 3 additional "random" bits, so the reading must be performed correctly.

On top of that, ESP32 is a little-endian chip, which means that the least significant byte of `uint16_t` and `uint32_t` variables is stored at the smallest address. Hence, if `uint16_t` is stored in memory, bits [7:0] are sent first, followed by bits [15:8].

For cases when the data to be transmitted has the size differing from `uint8_t` arrays, the following macros can be used to transform data to the format that can be sent by the SPI driver directly:

- *SPI_SWAP_DATA_TX* for data to be transmitted
- *SPI_SWAP_DATA_RX* for data received

Notes on Sending Mixed Transactions to the Same Device To reduce coding complexity, send only one type of transactions (interrupt or polling) to one Device. However, you still can send both interrupt and polling transactions alternately. The notes below explain how to do this.

The polling transactions should be initiated only after all the polling and interrupt transactions are finished.

Since an unfinished polling transaction blocks other transactions, please do not forget to call the function *spi_device_polling_end()* after *spi_device_polling_start()* to allow other transactions or to allow other Devices to use the bus. Remember that if there is no need to switch to other tasks during your polling transaction, you can initiate a transaction with *spi_device_polling_transmit()* so that it will be ended automatically.

In-flight polling transactions are disturbed by the ISR operation to accommodate interrupt transactions. Always make sure that all the interrupt transactions sent to the ISR are finished before you call *spi_device_polling_start()*. To do that, you can keep calling *spi_device_get_trans_result()* until all the transactions are returned.

To have better control of the calling sequence of functions, send mixed transactions to the same Device only within a single task.

Notes on Using the SPI Master driver on SPI1 Bus

Note: Though the *SPI Bus Lock* feature makes it possible to use SPI Master driver on the SPI1 bus, it's still tricky and needs a lot of special treatment. It's a feature for advanced developers.

To use SPI Master driver on SPI1 bus, you have to take care of two problems:

1. The code and data, required at the meanwhile the driver is operating SPI1 bus, should be in the internal memory. SPI1 bus is shared among devices and the cache for data (code) in the Flash as well as the PSRAM. The cache should be disabled during the other drivers are operating the SPI1 bus. Hence the data (code) in the flash as well as the PSRAM cannot be fetched at the meanwhile the driver acquires the SPI1 bus by:
   - Explicit bus acquiring between *spi_device_acquire_bus()* and *spi_device_release_bus()*.

- Implicit bus acquiring between `spi_device_polling_start()` and `spi_device_polling_end()` (or inside `spi_device_polling_transmit()`).

During the time above, all other tasks and most ISRs will be disabled (see *IRAM-Safe Interrupt Handlers*). Application code and data used by current task should be placed in internal memory (DRAM or IRAM), or already in the ROM. Access to external memory (flash code, const data in the flash, and static/heap data in the PSRAM) will cause a *Cache disabled but cached memory region accessed* exception. For differences between IRAM, DRAM, and flash cache, please refer to the *application memory layout* documentation.

To place functions into the IRAM, you can either:

1. Add *IRAM_ATTR* (include "esp_attr.h") to the function like:
   IRAM_ATTR void foo(void) { }
   Please note that when a function is inlined, it will follow its caller's segment, and the attribute will not take effect. You may need to use *NOLINE_ATTR* to avoid this.
2. Use the *noflash* placement in the *linker.lf*. See more in *Linker Script Generation*. Please note that, some code may be transformed into lookup table in the const data by the compiler, so *noflash_text* is not safe.

Please do take care that the optimization level may affect the compiler behavior of inline, or transforming some code into lookup table in the const data, etc.

To place data into the DRAM, you can either:

1. Add *DRAM_ATTR* (include "esp_attr.h") to the data definition like:
   DRAM_ATTR int g_foo = 3;
2. Use the *noflash* placement in the linker.lf. See more in *Linker Script Generation*.

Please also see the example peripherals/spi_master/hd_eeprom.

GPIO Matrix and IO_MUX Most of ESP32's peripheral signals have direct connection to their dedicated IO_MUX pins. However, the signals can also be routed to any other available pins using the less direct GPIO matrix. If at least one signal is routed through the GPIO matrix, then all signals will be routed through it.

The GPIO matrix introduces flexibility of routing but also brings the following disadvantages:

- Increases the input delay of the MISO signal, which makes MISO setup time violations more likely. If SPI needs to operate at high speeds, use dedicated IO_MUX pins.
- Allows signals with clock frequencies only up to 40 MHz, as opposed to 80 MHz if IO_MUX pins are used.

Note: For more details about the influence of the MISO input delay on the maximum clock frequency, see *Timing Considerations*.

The IO_MUX pins for SPI buses are given below.

Pin Name	SPI2	SPI3
	GPIO Number	
CS0*	15	5
SCLK	14	18
MISO	12	19
MOSI	13	23
QUADWP	2	22
QUADHD	4	21

* Only the first Device attached to the bus can use the CS0 pin.

Transfer Speed Considerations

There are three factors limiting the transfer speed:

- Transaction interval
- SPI clock frequency
- Cache miss of SPI functions, including callbacks

The main parameter that determines the transfer speed for large transactions is clock frequency. For multiple small transactions, the transfer speed is mostly determined by the length of transaction intervals.

Transaction Duration Transaction duration includes setting up SPI peripheral registers, copying data to FIFOs or setting up DMA links, and the time for SPI transaction.

Interrupt transactions allow appending extra overhead to accommodate the cost of FreeRTOS queues and the time needed for switching between tasks and the ISR.

For **interrupt transactions**, the CPU can switch to other tasks when a transaction is in progress. This saves the CPU time but increases the transaction duration. See *Interrupt Transactions*. For **polling transactions**, it does not block the task but allows to do polling when the transaction is in progress. For more information, see *Polling Transactions*.

If DMA is enabled, setting up the linked list requires about 2 us per transaction. When a master is transferring data, it automatically reads the data from the linked list. If DMA is not enabled, the CPU has to write and read each byte from the FIFO by itself. Usually, this is faster than 2 us, but the transaction length is limited to 64 bytes for both write and read.

Typical transaction duration for one byte of data are given below.

- Interrupt Transaction via DMA: 28 µs.
- Interrupt Transaction via CPU: 25 µs.
- Polling Transaction via DMA: 10 µs.
- Polling Transaction via CPU: 8 µs.

SPI Clock Frequency Transferring each byte takes eight times the clock period *8/fspi*.

If the clock frequency is too high, the use of some functions might be limited. See *Timing Considerations*.

Cache Miss The default config puts only the ISR into the IRAM. Other SPI related functions, including the driver itself and the callback, might suffer from cache misses and will need to wait until the code is read from flash. Select *CONFIG_SPI_MASTER_IN_IRAM* to put the whole SPI driver into IRAM and put the entire callback(s) and its callee functions into IRAM to prevent cache misses.

For an interrupt transaction, the overall cost is *20+8n/Fspi[MHz]* [us] for n bytes transferred in one transaction. Hence, the transferring speed is: *n/(20+8n/Fspi)*. An example of transferring speed at 8 MHz clock speed is given in the following table.

Frequency (MHz)	Transaction Interval (us)	Transaction Length (bytes)	Total Time (us)	Total Speed (KBps)
8	25	1	26	38.5
8	25	8	33	242.4
8	25	16	41	490.2
8	25	64	89	719.1
8	25	128	153	836.6

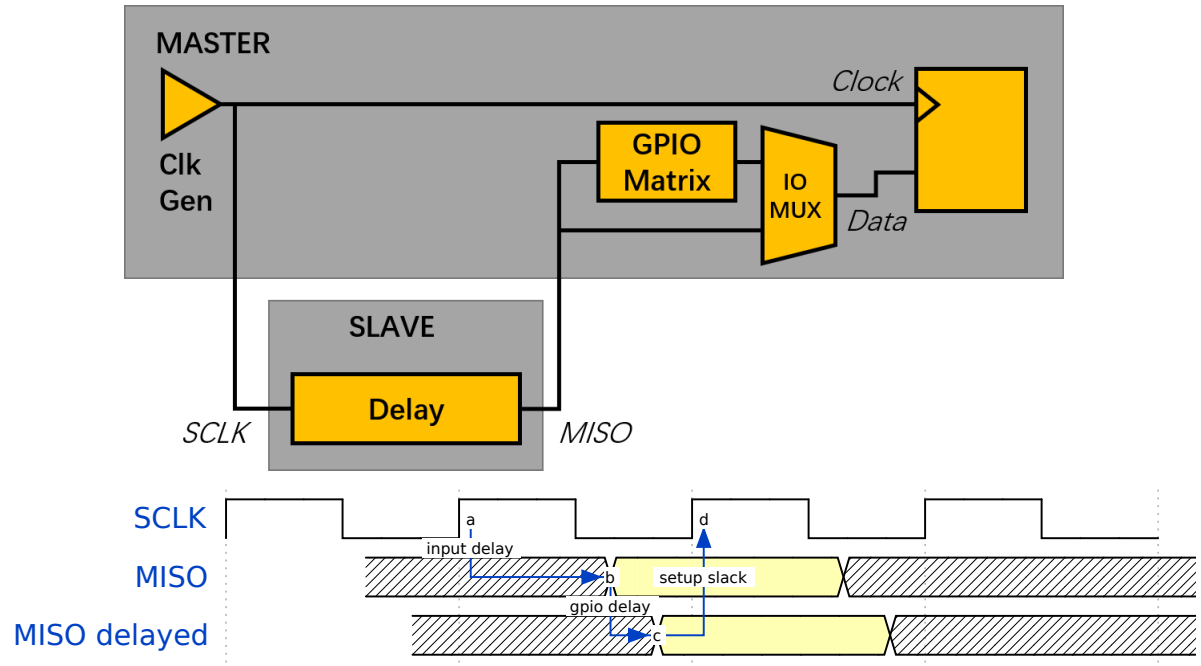
When a transaction length is short, the cost of transaction interval is high. If possible, try to squash several short transactions into one transaction to achieve a higher transfer speed.

Please note that the ISR is disabled during flash operation by default. To keep sending transactions during flash operations, enable *CONFIG_SPI_MASTER_ISR_IN_IRAM* and set `ESP_INTR_FLAG_IRAM` in the member `spi_bus_config_t::intr_flags`. In this case, all the transactions queued before starting flash operations will be handled by the ISR in parallel. Also note that the callback of each Device and their callee functions should be in IRAM, or your callback will crash due to cache miss. For more details, see *IRAM-Safe Interrupt Handlers*.

Timing Considerations

As shown in the figure below, there is a delay on the MISO line after the SCLK launch edge and before the signal is latched by the internal register. As a result, the MISO pin setup time is the limiting factor for the SPI clock speed.

When the delay is too long, the setup slack is < 0, which means the setup timing requirement is violated and the reading might be incorrect.



The maximum allowed frequency is dependent on:

- `spi_device_interface_config_t::input_delay_ns` - maximum data valid time on the MISO bus after a clock cycle on SCLK starts
- If the IO_MUX pin or the GPIO Matrix is used

When the GPIO matrix is used, the maximum allowed frequency is reduced to about 33~77% in comparison to the existing *input delay*. To retain a higher frequency, you have to use the IO_MUX pins or the *dummy bit workaround*. You can obtain the maximum reading frequency of the master by using the function `spi_get_freq_limit()`.

Dummy bit workaround: Dummy clocks, during which the Host does not read data, can be inserted before the read phase begins. The Device still sees the dummy clocks and sends out data, but the Host does not read until the read phase comes. This compensates for the lack of the MISO setup time required by the Host and allows the Host to do reading at a higher frequency.

In the ideal case, if the Device is so fast that the input delay is shorter than an APB clock cycle - 12.5 ns - the maximum frequency at which the Host can read (or read and write) in different conditions is as follows:

Frequency Limit (MHz)		Dummy Bits Used By Driver	Comments
GPIO matrix	IO_MUX pins		
26.6	80	No	
40	–	Yes	Half-duplex, no DMA allowed

If the Host only writes data, the *dummy bit workaround* and the frequency check can be disabled by setting the bit *SPI_DEVICE_NO_DUMMY* in the member `spi_device_interface_config_t::flags`. When disabled, the output frequency can be 80MHz, even if the GPIO matrix is used.

`spi_device_interface_config_t::flags`

The SPI master driver still works even if the `spi_device_interface_config_t::input_delay_ns` in the structure `spi_device_interface_config_t` is set to 0. However, setting an accurate value helps to:

- Calculate the frequency limit for full-duplex transactions
- Compensate the timing correctly with dummy bits for half-duplex transactions

You can approximate the maximum data valid time after the launch edge of SPI clocks by checking the statistics in the AC characteristics chapter of your Device's specification or measure the time using an oscilloscope or logic analyzer.

Please note that the actual PCB layout design and excessive loads may increase the input delay. It means that non-optimal wiring and/or a load capacitor on the bus will most likely lead to input delay values exceeding the values given in the Device specification or measured while the bus is floating.

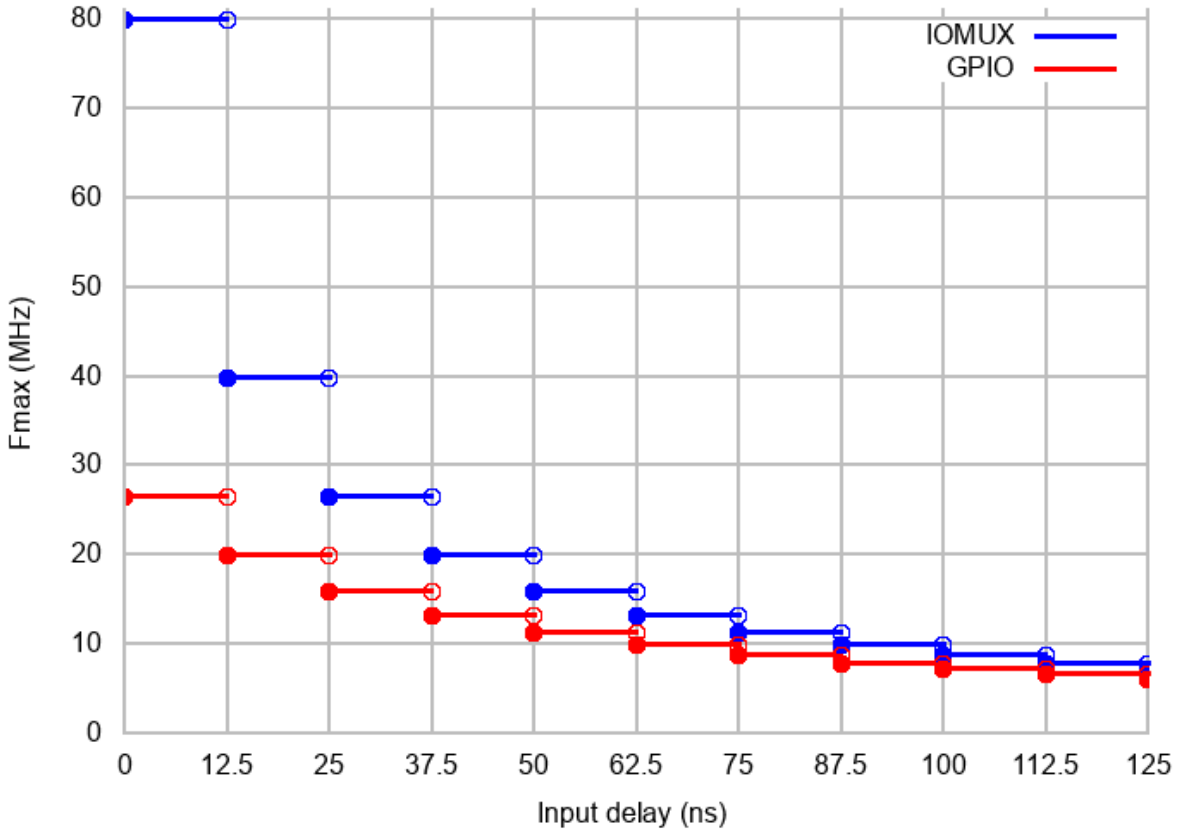
Some typical delay values are shown in the following table. (These data are retrieved when the slave device is on a different physical chip)

Device	Input delay (ns)
Ideal Device	0
ESP32 slave using IO_MUX*	50
ESP32 slave using GPIO_MUX*	75

The MISO path delay (valid time) consists of a slave's *input delay* plus master's *GPIO matrix delay*. This delay determines the frequency limit above which full-duplex transfers will not work as well as the dummy bits used in the half-duplex transactions. The frequency limit is:

Freq limit [MHz] = 80 / (floor(MISO delay[ns]/12.5) + 1)

The figure below shows the relationship between frequency limit and input delay. Two extra APB clock cycle periods should be added to the MISO delay if the master uses the GPIO matrix.



Corresponding frequency limits for different Devices with different *input delay* times are shown in the table below.

Master	Input delay (ns)	MISO path delay (ns)	Freq. limit (MHz)
IO_MUX (0ns)	0	0	80
	50	50	16
	75	75	11.43
GPIO (25ns)	0	25	26.67
	50	75	11.43
	75	100	8.89

Known Issues

1. Half-duplex transactions are not compatible with DMA when both writing and reading phases are used.
   If such transactions are required, you have to use one of the alternative solutions:
   1. Use full-duplex transactions instead.
   2. **Disable DMA by setting the bus initialization function's last parameter to 0 as follows:**
      ```
      ret=spi_bus_initialize(VSPI_HOST, &buscfg, 0);
      ```
   This can prohibit you from transmitting and receiving data longer than 64 bytes. 3. Try using the command and address fields to replace the write phase.
2. Full-duplex transactions are not compatible with the *dummy bit workaround*, hence the frequency is limited. See *dummy bit speed-up workaround*.
3. `dummy_bits` in *`spi_device_interface_config_t`* and *`spi_transaction_ext_t`* are not available when SPI read and write phases are both enabled (regardless of full duplex or half duplex mode).
4. `cs_ena_pretrans` is not compatible with the command and address phases of full-duplex transactions.

Application Example

The code example for using the SPI master half duplex mode to read/write a AT93C46D EEPROM (8-bit mode) can be found in the peripherals/spi_master/hd_eeprom directory of ESP-IDF examples.

API Reference - SPI Common

Header File

- components/hal/include/hal/spi_types.h

Structures

struct **`spi_line_mode_t`**

Line mode of SPI transaction phases: CMD, ADDR, DOUT/DIN.

Public Members

uint8_t **`cmd_lines`**

The line width of command phase, e.g. 2-line-cmd-phase.

uint8_t **`addr_lines`**

The line width of address phase, e.g. 1-line-addr-phase.

uint8_t **`data_lines`**

The line width of data phase, e.g. 4-line-data-phase.

Enumerations

enum **`spi_host_device_t`**

Enum with the three SPI peripherals that are software-accessible in it.

Values:

enumerator **`SPI1_HOST`**

SPI1.

enumerator **SPI2_HOST**

SPI2.

enumerator **SPI3_HOST**

SPI3.

enumerator **SPI_HOST_MAX**

invalid host value

enum **spi_clock_source_t**

Values:

enumerator **SPI_CLK_APB**

Select APB as the source clock.

enumerator **SPI_CLK_XTAL**

Select XTAL as the source clock.

enum **spi_event_t**

SPI Events.

Values:

enumerator **SPI_EV_BUF_TX**

The buffer has sent data to master.

enumerator **SPI_EV_BUF_RX**

The buffer has received data from master.

enumerator **SPI_EV_SEND_DMA_READY**

Slave has loaded its TX data buffer to the hardware (DMA).

enumerator **SPI_EV_SEND**

Master has received certain number of the data, the number is determined by Master.

enumerator **SPI_EV_RECV_DMA_READY**

Slave has loaded its RX data buffer to the hardware (DMA).

enumerator **SPI_EV_RECV**

Slave has received certain number of data from master, the number is determined by Master.

enumerator **SPI_EV_CMD9**

Received CMD9 from master.

enumerator **SPI_EV_CMDA**

Received CMDA from master.

enumerator **SPI_EV_TRANS**

A transaction has done.

enum **spi_command_t**

SPI command.

Values:

enumerator **SPI_CMD_HD_WRBUF**

enumerator **SPI_CMD_HD_RDBUF**

enumerator **SPI_CMD_HD_WRDMA**

enumerator **SPI_CMD_HD_RDDMA**

enumerator **SPI_CMD_HD_SEG_END**

enumerator **SPI_CMD_HD_EN_QPI**

enumerator **SPI_CMD_HD_WR_END**

enumerator **SPI_CMD_HD_INT0**

enumerator **SPI_CMD_HD_INT1**

enumerator **SPI_CMD_HD_INT2**

Header File

- components/driver/include/driver/spi_common.h

Functions

esp_err_t **spi_bus_initialize**(*spi_host_device_t* host_id, const *spi_bus_config_t* *bus_config, *spi_dma_chan_t* dma_chan)

Initialize a SPI bus.

Warning: SPI0/1 is not supported

Warning: If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Warning: The ISR of SPI is always executed on the core which calls this function. Never starve the ISR on this core or the SPI transactions will not be handled.

Parameters

- **host_id** –SPI peripheral that controls this bus
- **bus_config** –Pointer to a *spi_bus_config_t* struct specifying how the host should be initialized

- **dma_chan** –- Selecting a DMA channel for an SPI bus allows transactions on the bus with size only limited by the amount of internal memory.
  - Selecting SPI_DMA_DISABLED limits the size of transactions.
  - Set to SPI_DMA_DISABLED if only the SPI flash uses this bus.
  - Set to SPI_DMA_CH_AUTO to let the driver to allocate the DMA channel.

Returns

- ESP_ERR_INVALID_ARG if configuration is invalid
- ESP_ERR_INVALID_STATE if host already is in use
- ESP_ERR_NOT_FOUND if there is no available DMA channel
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **spi_bus_free**(*spi_host_device_t* host_id)

Free a SPI bus.

Warning: In order for this to succeed, all devices have to be removed first.

Parameters **host_id** –SPI peripheral to free

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if bus hasn't been initialized before, or not all devices on the bus are freed
- ESP_OK on success

Structures

struct **spi_bus_config_t**

This is a configuration structure for a SPI bus.

You can use this structure to specify the GPIO pins of the bus. Normally, the driver will use the GPIO matrix to route the signals. An exception is made when all signals either can be routed through the IO_MUX or are -1. In that case, the IO_MUX is used, allowing for >40MHz speeds.

Note: Be advised that the slave driver does not use the quadwp/quadhd lines and fields in *spi_bus_config_t* refering to these lines will be ignored and can thus safely be left uninitialized.

Public Members

int **mosi_io_num**

GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.

int **data0_io_num**

GPIO pin for spi data0 signal in quad/octal mode, or -1 if not used.

int **miso_io_num**

GPIO pin for Master In Slave Out (=spi_q) signal, or -1 if not used.

int **data1_io_num**

GPIO pin for spi data1 signal in quad/octal mode, or -1 if not used.

int **sclk_io_num**

GPIO pin for SPI Clock signal, or -1 if not used.

int **quadwp_io_num**

GPIO pin for WP (Write Protect) signal, or -1 if not used.

int **data2_io_num**

GPIO pin for spi data2 signal in quad/octal mode, or -1 if not used.

int **quadhd_io_num**

GPIO pin for HD (Hold) signal, or -1 if not used.

int **data3_io_num**

GPIO pin for spi data3 signal in quad/octal mode, or -1 if not used.

int **data4_io_num**

GPIO pin for spi data4 signal in octal mode, or -1 if not used.

int **data5_io_num**

GPIO pin for spi data5 signal in octal mode, or -1 if not used.

int **data6_io_num**

GPIO pin for spi data6 signal in octal mode, or -1 if not used.

int **data7_io_num**

GPIO pin for spi data7 signal in octal mode, or -1 if not used.

int **max_transfer_sz**

Maximum transfer size, in bytes. Defaults to 4092 if 0 when DMA enabled, or to `SOC_SPI_MAXIMUM_BUFFER_SIZE` if DMA is disabled.

uint32_t **flags**

Abilities of bus to be checked by the driver. Or-ed value of `SPICOMMON_BUSFLAG_*` flags.

int **intr_flags**

Interrupt flag for the bus to set the priority, and IRAM attribute, see `esp_intr_alloc.h`. Note that the EDGE, INTRDISABLED attribute are ignored by the driver. Note that if ESP_INTR_FLAG_IRAM is set, ALL the callbacks of the driver, and their callee functions, should be put in the IRAM.

Macros

SPI_MAX_DMA_LEN

SPI_SWAP_DATA_TX(DATA, LEN)

Transform unsigned integer of length <= 32 bits to the format which can be sent by the SPI driver directly.

E.g. to send 9 bits of data, you can:

```
uint16_t data = SPI_SWAP_DATA_TX(0x145, 9);
```

Then points tx_buffer to `&data`.

Parameters

- **DATA** –Data to be sent, can be uint8_t, uint16_t or uint32_t.
- **LEN** –Length of data to be sent, since the SPI peripheral sends from the MSB, this helps to shift the data to the MSB.

SPI_SWAP_DATA_RX(DATA, LEN)

Transform received data of length <= 32 bits to the format of an unsigned integer.

E.g. to transform the data of 15 bits placed in a 4-byte array to integer:

```
uint16_t data = SPI_SWAP_DATA_RX(*(uint32_t*)t->rx_data, 15);
```

Parameters

- **DATA** –Data to be rearranged, can be uint8_t, uint16_t or uint32_t.
- **LEN** –Length of data received, since the SPI peripheral writes from the MSB, this helps to shift the data to the LSB.

SPICOMMON_BUSFLAG_SLAVE

Initialize I/O in slave mode.

SPICOMMON_BUSFLAG_MASTER

Initialize I/O in master mode.

SPICOMMON_BUSFLAG_IOMUX_PINS

Check using iomux pins. Or indicates the pins are configured through the IO mux rather than GPIO matrix.

SPICOMMON_BUSFLAG_GPIO_PINS

Force the signals to be routed through GPIO matrix. Or indicates the pins are routed through the GPIO matrix.

SPICOMMON_BUSFLAG_SCLK

Check existing of SCLK pin. Or indicates CLK line initialized.

SPICOMMON_BUSFLAG_MISO

Check existing of MISO pin. Or indicates MISO line initialized.

SPICOMMON_BUSFLAG_MOSI

Check existing of MOSI pin. Or indicates MOSI line initialized.

SPICOMMON_BUSFLAG_DUAL

Check MOSI and MISO pins can output. Or indicates bus able to work under DIO mode.

SPICOMMON_BUSFLAG_WPHD

Check existing of WP and HD pins. Or indicates WP & HD pins initialized.

SPICOMMON_BUSFLAG_QUAD

Check existing of MOSI/MISO/WP/HD pins as output. Or indicates bus able to work under QIO mode.

SPICOMMON_BUSFLAG_IO4_IO7

Check existing of IO4~IO7 pins. Or indicates IO4~IO7 pins initialized.

SPICOMMON_BUSFLAG_OCTAL

Check existing of MOSI/MISO/WP/HD/SPIIO4/SPIIO5/SPIIO6/SPIIO7 pins as output. Or indicates bus able to work under octal mode.

SPICOMMON_BUSFLAG_NATIVE_PINS

Type Definitions

typedef *spi_common_dma_t* **spi_dma_chan_t**

Enumerations

enum **spi_common_dma_t**

SPI DMA channels.

Values:

enumerator **SPI_DMA_DISABLED**

Do not enable DMA for SPI.

enumerator **SPI_DMA_CH1**

Enable DMA, select DMA Channel 1.

enumerator **SPI_DMA_CH2**

Enable DMA, select DMA Channel 2.

enumerator **SPI_DMA_CH_AUTO**

Enable DMA, channel is automatically selected by driver.

API Reference - SPI Master

Header File

- components/driver/include/driver/spi_master.h

Functions

esp_err_t **spi_bus_add_device**(*spi_host_device_t* host_id, const *spi_device_interface_config_t* *dev_config, *spi_device_handle_t* *handle)

Allocate a device on a SPI bus.

This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control up to three devices.

Note: While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.

Parameters

- **host_id** –SPI peripheral to allocate device on
- **dev_config** –SPI interface protocol config for the device
- **handle** –Pointer to variable to hold the device handle

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND if host doesn't have any free CS slots
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **spi_bus_remove_device** (*spi_device_handle_t* handle)

Remove a device from the SPI bus.

Parameters **handle** –Device handle to free

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if device already is freed
- ESP_OK on success

esp_err_t **spi_device_queue_trans** (*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc, TickType_t ticks_to_wait)

Queue a SPI transaction for interrupt transaction execution. Get the result by `spi_device_get_trans_result`.

Note: Normally a device cannot start (queue) polling and interrupt transactions simultaneously.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **trans_desc** –Description of transaction to execute
- **ticks_to_wait** –Ticks to wait until there's room in the queue; use port-MAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid. This can happen if SPI_TRANS_CS_KEEP_ACTIVE flag is specified while the bus was not acquired (`spi_device_acquire_bus()` should be called first)
- ESP_ERR_TIMEOUT if there was no room in the queue before ticks_to_wait expired
- ESP_ERR_NO_MEM if allocating DMA-capable temporary buffer failed
- ESP_ERR_INVALID_STATE if previous transactions are not finished
- ESP_OK on success

esp_err_t **spi_device_get_trans_result** (*spi_device_handle_t* handle, *spi_transaction_t* **trans_desc, TickType_t ticks_to_wait)

Get the result of a SPI transaction queued earlier by `spi_device_queue_trans`.

This routine will wait until a transaction to the given device succesfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **trans_desc** –Pointer to variable able to contain a pointer to the description of the transaction that is executed. The descriptor should not be modified until the descriptor is returned by spi_device_get_trans_result.
- **ticks_to_wait** –Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if there was no completed transaction before ticks_to_wait expired
- ESP_OK on success

esp_err_t **spi_device_transmit** (*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc)

Send a SPI transaction, wait for it to complete, and return the result.

This function is the equivalent of calling spi_device_queue_trans() followed by spi_device_get_trans_result(). Do not use this when there is still a transaction separately queued (started) from spi_device_queue_trans() or polling_start/transmit that hasn't been finalized.

Note: This function is not thread safe when multiple tasks access the same SPI device. Normally a device cannot start (queue) polling and interrupt transactions simutanuously.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **trans_desc** –Description of transaction to execute

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **spi_device_polling_start**(*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc, TickType_t ticks_to_wait)

Immediately start a polling transaction.

Note: Normally a device cannot start (queue) polling and interrupt transactions simutanuously. Moreover, a device cannot start a new polling transaction if another polling transaction is not finished.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **trans_desc** –Description of transaction to execute
- **ticks_to_wait** –Ticks to wait until there's room in the queue; currently only portMAX_DELAY is supported.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid. This can happen if SPI_TRANS_CS_KEEP_ACTIVE flag is specified while the bus was not acquired (`spi_device_acquire_bus()` should be called first)
- ESP_ERR_TIMEOUT if the device cannot get control of the bus before `ticks_to_wait` expired
- ESP_ERR_NO_MEM if allocating DMA-capable temporary buffer failed
- ESP_ERR_INVALID_STATE if previous transactions are not finished
- ESP_OK on success

esp_err_t **spi_device_polling_end**(*spi_device_handle_t* handle, TickType_t ticks_to_wait)

Poll until the polling transaction ends.

This routine will not return until the transaction to the given device has succesfully completed. The task is not blocked, but actively busy-spins for the transaction to be completed.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **ticks_to_wait** –Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if the transaction cannot finish before ticks_to_wait expired
- ESP_OK on success

esp_err_t **spi_device_polling_transmit**(*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc)

Send a polling transaction, wait for it to complete, and return the result.

This function is the equivalent of calling spi_device_polling_start() followed by spi_device_polling_end(). Do not use this when there is still a transaction that hasn't been finalized.

Note: This function is not thread safe when multiple tasks access the same SPI device. Normally a device

cannot start (queue) polling and interrupt transactions simutanuously.

Parameters

- **handle** –Device handle obtained using spi_host_add_dev
- **trans_desc** –Description of transaction to execute

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if the device cannot get control of the bus
- ESP_ERR_NO_MEM if allocating DMA-capable temporary buffer failed
- ESP_ERR_INVALID_STATE if previous transactions of same device are not finished
- ESP_OK on success

esp_err_t **spi_device_acquire_bus** (*spi_device_handle_t* device, TickType_t wait)

Occupy the SPI bus for a device to do continuous transactions.

Transactions to all other devices will be put off until `spi_device_release_bus` is called.

Note: The function will wait until all the existing transactions have been sent.

Parameters

- **device** –The device to occupy the bus.
- **wait** –Time to wait before the the bus is occupied by the device. Currently MUST set to portMAX_DELAY.

Returns

- ESP_ERR_INVALID_ARG : `wait` is not set to portMAX_DELAY.
- ESP_OK : Success.

void **spi_device_release_bus** (*spi_device_handle_t* dev)

Release the SPI bus occupied by the device. All other devices can start sending transactions.

Parameters **dev** –The device to release the bus.

int **spi_get_actual_clock** (int fapb, int hz, int duty_cycle)

Calculate the working frequency that is most close to desired frequency.

Parameters

- **fapb** –The frequency of apb clock, should be `APB_CLK_FREQ`.
- **hz** –Desired working frequency
- **duty_cycle** –Duty cycle of the spi clock

Returns Actual working frequency that most fit.

void **spi_get_timing** (bool gpio_is_used, int input_delay_ns, int eff_clk, int *dummy_o, int *cycles_remain_o)

Calculate the timing settings of specified frequency and settings.

Note: If **dummy_o* is not zero, it means dummy bits should be applied in half duplex mode, and full duplex mode may not work.

Parameters

- **gpio_is_used** –True if using GPIO matrix, or False if iomux pins are used.
- **input_delay_ns** –Input delay from SCLK launch edge to MISO data valid.
- **eff_clk** –Effective clock frequency (in Hz) from `spi_get_actual_clock()`.
- **dummy_o** –Address of dummy bits used output. Set to NULL if not needed.
- **cycles_remain_o** –Address of cycles remaining (after dummy bits are used) output.
  - -1 If too many cycles remaining, suggest to compensate half a clock.
  - 0 If no remaining cycles or dummy bits are not used.

- positive value: cycles suggest to compensate.

int **spi_get_freq_limit**(bool gpio_is_used, int input_delay_ns)

Get the frequency limit of current configurations. SPI master working at this limit is OK, while above the limit, full duplex mode and DMA will not work, and dummy bits will be aplied in the half duplex mode.

Parameters
- **gpio_is_used** –True if using GPIO matrix, or False if native pins are used.
- **input_delay_ns** –Input delay from SCLK launch edge to MISO data valid.

Returns Frequency limit of current configurations.

esp_err_t **spi_bus_get_max_transaction_len**(*spi_host_device_t* host_id, size_t *max_bytes)

Get max length (in bytes) of one transaction.

Parameters
- **host_id** –SPI peripheral
- **max_bytes** –**[out]** Max length of one transaction, in bytes

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument

Structures

struct **spi_device_interface_config_t**

This is a configuration for a SPI slave device that is connected to one of the SPI buses.

Public Members

uint8_t **command_bits**

Default amount of bits in command phase (0-16), used when `SPI_TRANS_VARIABLE_CMD` is not used, otherwise ignored.

uint8_t **address_bits**

Default amount of bits in address phase (0-64), used when `SPI_TRANS_VARIABLE_ADDR` is not used, otherwise ignored.

uint8_t **dummy_bits**

Amount of dummy bits to insert between address and data phase.

uint8_t **mode**

SPI mode, representing a pair of (CPOL, CPHA) configuration:

- 0: (0, 0)
- 1: (0, 1)
- 2: (1, 0)
- 3: (1, 1)

uint16_t **duty_cycle_pos**

Duty cycle of positive clock, in 1/256th increments (128 = 50%/50% duty). Setting this to 0 (=not setting it) is equivalent to setting this to 128.

uint16_t **cs_ena_pretrans**

Amount of SPI bit-cycles the cs should be activated before the transmission (0-16). This only works on half-duplex transactions.

uint8_t **cs_ena_posttrans**

Amount of SPI bit-cycles the cs should stay active after the transmission (0-16)

int **clock_speed_hz**

Clock speed, divisors of 80MHz, in Hz. See `SPI_MASTER_FREQ_*`.

int **input_delay_ns**

Maximum data valid time of slave. The time required between SCLK and MISO valid, including the possible clock delay from slave to master. The driver uses this value to give an extra delay before the MISO is ready on the line. Leave at 0 unless you know you need a delay. For better timing performance at high frequency (over 8MHz), it' s suggest to have the right value.

int **spics_io_num**

CS GPIO pin for this device, or -1 if not used.

uint32_t **flags**

Bitwise OR of SPI_DEVICE_* flags.

int **queue_size**

Transaction queue size. This sets how many transactions can be 'in the air' (queued using spi_device_queue_trans but not yet finished using spi_device_get_trans_result) at the same time.

transaction_cb_t **pre_cb**

Callback to be called before a transmission is started.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

transaction_cb_t **post_cb**

Callback to be called after a transmission has completed.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

struct **spi_transaction_t**

This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.

Public Members

uint32_t **flags**

Bitwise OR of SPI_TRANS_* flags.

uint16_t **cmd**

Command data, of which the length is set in the `command_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be "command" in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF 3.0.

Example: write 0x0123 and command_bits=12 to send command 0x12, 0x3_ (in previous version, you may have to write 0x3_12).

uint64_t **addr**

Address data, of which the length is set in the `address_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be "address" in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF3.0.

Example: write 0x123400 and address_bits=24 to send address of 0x12, 0x34, 0x00 (in previous version, you may have to write 0x12340000).

size_t **length**

Total data length, in bits.

size_t **rxlength**

Total data length received, should be not greater than `length` in full-duplex mode (0 defaults this to the value of `length`).

void ***user**

User-defined variable. Can be used to store eg transaction ID.

const void ***tx_buffer**

Pointer to transmit buffer, or NULL for no MOSI phase.

uint8_t **tx_data**[4]

If SPI_TRANS_USE_TXDATA is set, data set here is sent directly from this variable.

void ***rx_buffer**

Pointer to receive buffer, or NULL for no MISO phase. Written by 4 bytes-unit if DMA is used.

uint8_t **rx_data**[4]

If SPI_TRANS_USE_RXDATA is set, data is received directly to this variable.

struct **spi_transaction_ext_t**

This struct is for SPI transactions which may change their address and command length. Please do set the flags in base to `SPI_TRANS_VARIABLE_CMD_ADR` to use the bit length here.

Public Members

struct *spi_transaction_t* **base**

Transaction data, so that pointer to *spi_transaction_t* can be converted into *spi_transaction_ext_t*.

uint8_t **command_bits**

The command length in this transaction, in bits.

uint8_t **address_bits**

The address length in this transaction, in bits.

uint8_t **dummy_bits**

The dummy length in this transaction, in bits.

Macros

SPI_MASTER_FREQ_8M

SPI master clock is divided by 80MHz apb clock. Below defines are example frequencies, and are accurate. Be free to specify a random frequency, it will be rounded to closest frequency (to macros below if above 8MHz). 8MHz

SPI_MASTER_FREQ_9M

8.89MHz

SPI_MASTER_FREQ_10M

10MHz

SPI_MASTER_FREQ_11M

11.43MHz

SPI_MASTER_FREQ_13M

13.33MHz

SPI_MASTER_FREQ_16M

16MHz

SPI_MASTER_FREQ_20M

20MHz

SPI_MASTER_FREQ_26M

26.67MHz

SPI_MASTER_FREQ_40M

40MHz

SPI_MASTER_FREQ_80M

80MHz

SPI_DEVICE_TXBIT_LSBFIRST

Transmit command/address/data LSB first instead of the default MSB first.

SPI_DEVICE_RXBIT_LSBFIRST

Receive data LSB first instead of the default MSB first.

SPI_DEVICE_BIT_LSBFIRST

Transmit and receive LSB first.

SPI_DEVICE_3WIRE

Use MOSI (=spid) for both sending and receiving data.

SPI_DEVICE_POSITIVE_CS

Make CS positive during a transaction instead of negative.

SPI_DEVICE_HALFDUPLEX

Transmit data before receiving it, instead of simultaneously.

SPI_DEVICE_CLK_AS_CS

Output clock on CS line if CS is active.

SPI_DEVICE_NO_DUMMY

There are timing issue when reading at high frequency (the frequency is related to whether iomux pins are used, valid time after slave sees the clock).

- In half-duplex mode, the driver automatically inserts dummy bits before reading phase to fix the timing issue. Set this flag to disable this feature.
- In full-duplex mode, however, the hardware cannot use dummy bits, so there is no way to prevent data being read from getting corrupted. Set this flag to confirm that you're going to work with output only, or read without dummy bits at your own risk.

SPI_DEVICE_DDRCLK

SPI_TRANS_MODE_DIO

Transmit/receive data in 2-bit mode.

SPI_TRANS_MODE_QIO

Transmit/receive data in 4-bit mode.

SPI_TRANS_USE_RXDATA

Receive into rx_data member of *spi_transaction_t* instead into memory at rx_buffer.

SPI_TRANS_USE_TXDATA

Transmit tx_data member of *spi_transaction_t* instead of data at tx_buffer. Do not set tx_buffer when using this.

SPI_TRANS_MODE_DIOQIO_ADDR

Also transmit address in mode selected by SPI_MODE_DIO/SPI_MODE_QIO.

SPI_TRANS_VARIABLE_CMD

Use the `command_bits` in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_VARIABLE_ADDR

Use the `address_bits` in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_VARIABLE_DUMMY

Use the `dummy_bits` in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_CS_KEEP_ACTIVE

Keep CS active after data transfer.

SPI_TRANS_MULTILINE_CMD

The data lines used at command phase is the same as data phase (otherwise, only one data line is used at command phase)

SPI_TRANS_MODE_OCT

Transmit/receive data in 8-bit mode.

SPI_TRANS_MULTILINE_ADDR

The data lines used at address phase is the same as data phase (otherwise, only one data line is used at address phase)

Type Definitions

typedef struct *spi_transaction_t* **spi_transaction_t**

typedef void (***transaction_cb_t**)(*spi_transaction_t* *trans)

typedef struct spi_device_t ***spi_device_handle_t**

Handle for a device on a SPI bus.

2.6.21 SPI Slave Driver

SPI Slave driver is a program that controls ESP32's SPI peripherals while they function as slaves.

Overview of ESP32's SPI peripherals

ESP32 integrates two general purpose SPI controllers which can be used as slave nodes driven by an off-chip SPI master.

- SPI2, sometimes referred to as HSPI
- SPI3, sometimes referred to as VSPI

SPI2 and SPI3 have independent signal buses with the same respective names.

Terminology

The terms used in relation to the SPI slave driver are given in the table below.

Term	Definition
Host	The SPI controller peripheral external to ESP32 that initiates SPI transmissions over the bus, and acts as an SPI Master.
Device	SPI slave device (general purpose SPI controller). Each Device shares the MOSI, MISO and SCLK signals but is only active on the bus when the Host asserts the Device's individual CS line.
Bus	A signal bus, common to all Devices connected to one Host. In general, a bus includes the following lines: MISO, MOSI, SCLK, one or more CS lines, and, optionally, QUADWP and QUADHD. So Devices are connected to the same lines, with the exception that each Device has its own CS line. Several Devices can also share one CS line if connected in the daisy-chain manner.
MISO	Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host.
MOSI	Master Out, Slave In, a.k.a. D. Data transmission from a Host to Device.
SCLK	Serial Clock. Oscillating signal generated by a Host that keeps the transmission of data bits in sync.
CS	Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data.
QUADWP	Write Protect signal. Only used for 4-bit (qio/qout) transactions.
QUADHD	Hold signal. Only used for 4-bit (qio/qout) transactions.
Assertion	The action of activating a line. The opposite action of returning the line back to inactive (back to idle) is called *de-assertion*.
Transaction	One instance of a Host asserting a CS line, transferring data to and from a Device, and de-asserting the CS line. Transactions are atomic, which means they can never be interrupted by another transaction.
Launch Edge	Edge of the clock at which the source register *launches* the signal onto the line.
Latch Edge	Edge of the clock at which the destination register *latches in* the signal.

Driver Features

The SPI slave driver allows using the SPI peripherals as full-duplex Devices. The driver can send/receive transactions up to 64 bytes in length, or utilize DMA to send/receive longer transactions. However, there are some *known issues* related to DMA.

SPI Transactions

A full-duplex SPI transaction begins when the Host asserts the CS line and starts sending out clock pulses on the SCLK line. Every clock pulse, a data bit is shifted from the Host to the Device on the MOSI line and back on the MISO line at the same time. At the end of the transaction, the Host de-asserts the CS line.

The attributes of a transaction are determined by the configuration structure for an SPI peripheral acting as a slave device `spi_slave_interface_config_t`, and transaction configuration structure `spi_slave_transaction_t`.

As not every transaction requires both writing and reading data, you have a choice to configure the `spi_transaction_t` structure for TX only, RX only, or TX and RX transactions. If `spi_slave_transaction_t::rx_buffer` is set to NULL, the read phase will be skipped. If `spi_slave_transaction_t::tx_buffer` is set to NULL, the write phase will be skipped.

Note: A Host should not start a transaction before its Device is ready for receiving data. It is recommended to use another GPIO pin for a handshake signal to sync the Devices. For more details, see *Transaction Interval*.

Driver Usage

- Initialize an SPI peripheral as a Device by calling the function cpp:func:*spi_slave_initialize*. Make sure to set the correct I/O pins in the struct *bus_config*. Set the unused signals to `-1`.

If transactions will be longer than 32 bytes, allow a DMA channel 1 or 2 by setting the parameter `dma_chan` to `1` or `2` respectively. Otherwise, set `dma_chan` to `0`.

- Before initiating transactions, fill one or more *`spi_slave_transaction_t`* structs with the transaction parameters required. Either queue all transactions by calling the function *`spi_slave_queue_trans()`* and, at a later time, query the result by using the function *`spi_slave_get_trans_result()`*, or handle all requests individually by feeding them into *`spi_slave_transmit()`*. The latter two functions will be blocked until the Host has initiated and finished a transaction, causing the queued data to be sent and received.
- (Optional) To unload the SPI slave driver, call *`spi_slave_free()`*.

Transaction Data and Master/Slave Length Mismatches

Normally, the data that needs to be transferred to or from a Device is read or written to a chunk of memory indicated by the *`spi_slave_transaction_t::rx_buffer`* and *`spi_slave_transaction_t::tx_buffer`*. The SPI driver can be configured to use DMA for transfers, in which case these buffers must be allocated in DMA-capable memory using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.

The amount of data that the driver can read or write to the buffers is limited by *`spi_slave_transaction_t::length`*. However, this member does not define the actual length of an SPI transaction. A transaction's length is determined by the clock and CS lines driven by the Host. The actual length of the transmission can be read only after a transaction is finished from the member *`spi_slave_transaction_t::trans_len`*.

If the length of the transmission is greater than the buffer length, only the initial number of bits specified in the *`spi_slave_transaction_t::length`* member will be sent and received. In this case, *`spi_slave_transaction_t::trans_len`* is set to *`spi_slave_transaction_t::length`* instead of the actual transaction length. To meet the actual transaction length requirements, set *`spi_slave_transaction_t::length`* to a value greater than the maximum *`spi_slave_transaction_t::trans_len`* expected. If the transmission length is shorter than the buffer length, only the data equal to the length of the buffer will be transmitted.

GPIO Matrix and IO_MUX

Most of ESP32's peripheral signals have direct connection to their dedicated IO_MUX pins. However, the signals can also be routed to any other available pins using the less direct GPIO matrix.

If at least one signal is routed through the GPIO matrix, then all signals will be routed through it. The GPIO matrix samples all signals at 80 MHz and transmits them between the GPIO and the peripheral.

If the driver is configured so that all SPI signals are either routed to their dedicated IO_MUX pins or are not connected at all, the GPIO matrix will be bypassed.

The GPIO matrix introduces flexibility of routing but also increases the input delay of the MISO signal, which makes MISO setup time violations more likely. If SPI needs to operate at high speeds, use dedicated IO_MUX pins.

Note: For more details about the influence of the MISO input delay on the maximum clock frequency, see *Timing Considerations*.

The IO_MUX pins for SPI buses are given below.

Pin Name	GPIO Number (SPI2)	GPIO Number (SPI3)
CS0*	15	5
SCLK	14	18
MISO	12	19
MOSI	13	23
QUADWP	2	22
QUADHD	4	21

- Only the first Device attached to the bus can use the CS0 pin.

Speed and Timing Considerations

Transaction Interval The ESP32 SPI slave peripherals are designed as general purpose Devices controlled by a CPU. As opposed to dedicated slaves, CPU-based SPI Devices have a limited number of pre-defined registers. All transactions must be handled by the CPU, which means that the transfers and responses are not real-time, and there might be noticeable latency.

As a solution, a Device's response rate can be doubled by using the functions `spi_slave_queue_trans()` and then `spi_slave_get_trans_result()` instead of using `spi_slave_transmit()`.

You can also configure a GPIO pin through which the Device will signal to the Host when it is ready for a new transaction. A code example of this can be found in peripherals/spi_slave.

SCLK Frequency Requirements The SPI slaves are designed to operate at up to 10 MHz. The data cannot be recognized or received correctly if the clock is too fast or does not have a 50% duty cycle.

On top of that, there are additional requirements for the data to meet the timing constraints:

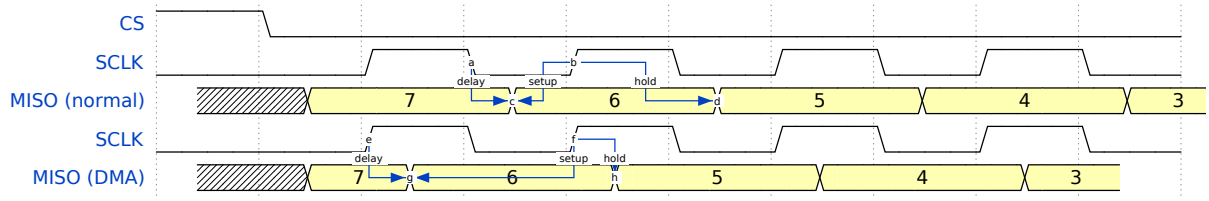
- **Read (MOSI):** The Device can read data correctly only if the data is already set at the launch edge. Although it is usually the case for most masters.
- **Write (MISO):** The output delay of the MISO signal needs to be shorter than half of a clock cycle period so that the MISO line is stable before the next latch edge. Given that the clock is balanced, the output delay and frequency limitations in different cases are given below.

/	Output delay of MISO (ns)	Freq. limit (MHz)
IO_MUX	43.75	<11.4
GPIO matrix	68.75	<7.2

Note: 1. If the frequency reaches the maximum limitation, random errors may occur. 2. The clock uncertainty between the Host and the Device (12.5 ns) is included. 3. The output delay is measured under ideal circumstances (no load). If the MISO pin is heavily loaded, the output delay will be longer, and the maximum allowed frequency will be lower.
Exception: The frequency is allowed to be higher if the master has more tolerance for the MISO setup time, e.g., latch data at the next edge, or configurable latching time.

Restrictions and Known Issues

1. If DMA is enabled, the rx buffer should be word-aligned (starting from a 32-bit boundary and having a length of multiples of 4 bytes). Otherwise, DMA may write incorrectly or not in a boundary aligned manner. The driver reports an error if this condition is not satisfied.
   Also, a Host should write lengths that are multiples of 4 bytes. The data with inappropriate lengths will be discarded.
2. Furthermore, DMA requires SPI modes 1 and 3. For SPI modes 0 and 2, the MISO signal has to be launched half a clock cycle earlier to meet the timing. The new timing is as follows:



If DMA is enabled, a Device's launch edge is half of an SPI clock cycle ahead of the normal time, shifting to the Master's actual latch edge. In this case, if the GPIO matrix is bypassed, the hold time for data sampling is 68.75 ns and no longer a half of an SPI clock cycle. If the GPIO matrix is used, the hold time will increase to 93.75 ns. The Host should sample the data immediately at the latch edge or communicate in SPI modes 1 or 3. If your Host cannot meet these timing requirements, initialize your Device without DMA.

Application Example

The code example for Device/Host communication can be found in the peripherals/spi_slave directory of ESP-IDF examples.

API Reference

Header File

- components/driver/include/driver/spi_slave.h

Functions

esp_err_t **spi_slave_initialize**(*spi_host_device_t* host, const *spi_bus_config_t* *bus_config, const *spi_slave_interface_config_t* *slave_config, *spi_dma_chan_t* dma_chan)

Initialize a SPI bus as a slave interface.

Warning: SPI0/1 is not supported

Warning: If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Warning: The ISR of SPI is always executed on the core which calls this function. Never starve the ISR on this core or the SPI transactions will not be handled.

Parameters

- **host** –SPI peripheral to use as a SPI slave interface
- **bus_config** –Pointer to a *spi_bus_config_t* struct specifying how the host should be initialized
- **slave_config** –Pointer to a *spi_slave_interface_config_t* struct specifying the details for the slave interface
- **dma_chan** –- Selecting a DMA channel for an SPI bus allows transactions on the bus with size only limited by the amount of internal memory.
  - Selecting SPI_DMA_DISABLED limits the size of transactions.
  - Set to SPI_DMA_DISABLED if only the SPI flash uses this bus.
  - Set to SPI_DMA_CH_AUTO to let the driver to allocate the DMA channel.

Returns

- ESP_ERR_INVALID_ARG if configuration is invalid
- ESP_ERR_INVALID_STATE if host already is in use

- ESP_ERR_NOT_FOUND if there is no available DMA channel
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

esp_err_t **spi_slave_free** (*spi_host_device_t* host)

Free a SPI bus claimed as a SPI slave interface.

Parameters **host** –SPI peripheral to free

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if not all devices on the bus are freed
- ESP_OK on success

esp_err_t **spi_slave_queue_trans** (*spi_host_device_t* host, const *spi_slave_transaction_t* *trans_desc, TickType_t ticks_to_wait)

Queue a SPI transaction for execution.

Queues a SPI transaction to be executed by this slave device. (The transaction queue size was specified when the slave device was initialised via spi_slave_initialize.) This function may block if the queue is full (depending on the ticks_to_wait parameter). No SPI operation is directly initiated by this function, the next queued transaction will happen when the master initiates a SPI transaction by pulling down CS and sending out clock signals.

This function hands over ownership of the buffers in `trans_desc` to the SPI slave driver; the application is not to access this memory until `spi_slave_queue_trans` is called to hand ownership back to the application.

Parameters

- **host** –SPI peripheral that is acting as a slave
- **trans_desc** –Description of transaction to execute. Not const because we may want to write status back into the transaction description.
- **ticks_to_wait** –Ticks to wait until there's room in the queue; use portMAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **spi_slave_get_trans_result** (*spi_host_device_t* host, *spi_slave_transaction_t* **trans_desc, TickType_t ticks_to_wait)

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with spi_slave_queue_trans) has succesfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

It is mandatory to eventually use this function for any transaction queued by `spi_slave_queue_trans`.

Parameters

- **host** –SPI peripheral to that is acting as a slave
- **trans_desc** –**[out]** Pointer to variable able to contain a pointer to the description of the transaction that is executed
- **ticks_to_wait** –Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

esp_err_t **spi_slave_transmit** (*spi_host_device_t* host, *spi_slave_transaction_t* *trans_desc, TickType_t ticks_to_wait)

Do a SPI transaction.

Essentially does the same as spi_slave_queue_trans followed by spi_slave_get_trans_result. Do not use this when there is still a transaction queued that hasn't been finalized using spi_slave_get_trans_result.

Parameters

- **host** –SPI peripheral to that is acting as a slave
- **trans_desc** –Pointer to variable able to contain a pointer to the description of the transaction that is executed. Not const because we may want to write status back into the transaction description.
- **ticks_to_wait** –Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Structures

struct **spi_slave_interface_config_t**

This is a configuration for a SPI host acting as a slave device.

Public Members

int **spics_io_num**

CS GPIO pin for this device.

uint32_t **flags**

Bitwise OR of SPI_SLAVE_* flags.

int **queue_size**

Transaction queue size. This sets how many transactions can be 'in the air' (queued using spi_slave_queue_trans but not yet finished using spi_slave_get_trans_result) at the same time.

uint8_t **mode**

SPI mode, representing a pair of (CPOL, CPHA) configuration:

- 0: (0, 0)
- 1: (0, 1)
- 2: (1, 0)
- 3: (1, 1)

slave_transaction_cb_t **post_setup_cb**

Callback called after the SPI registers are loaded with new data.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

slave_transaction_cb_t **post_trans_cb**

Callback called after a transaction is done.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

struct **spi_slave_transaction_t**

This structure describes one SPI transaction

Public Members

size_t **length**

Total data length, in bits.

size_t **trans_len**

Transaction data length, in bits.

const void ***tx_buffer**

Pointer to transmit buffer, or NULL for no MOSI phase.

void ***rx_buffer**

Pointer to receive buffer, or NULL for no MISO phase. When the DMA is anabled, must start at WORD boundary (`rx_buffer%4==0`), and has length of a multiple of 4 bytes.

void ***user**

User-defined variable. Can be used to store eg transaction ID.

Macros

SPI_SLAVE_TXBIT_LSBFIRST

Transmit command/address/data LSB first instead of the default MSB first.

SPI_SLAVE_RXBIT_LSBFIRST

Receive data LSB first instead of the default MSB first.

SPI_SLAVE_BIT_LSBFIRST

Transmit and receive LSB first.

Type Definitions

typedef struct *spi_slave_transaction_t* **spi_slave_transaction_t**

typedef void (***slave_transaction_cb_t**)(*spi_slave_transaction_t* *trans)

2.6.22 ESP32-WROOM-32SE (Secure Element)

Overview

The ESP32-WROOM-32SE has integrated Microchip ' s ATECC608A cryptoauth chip in the module. ATECC608A is secure element which would generate and store ECC private key in the hardware.The ECC private key can be used to enhance security to connect to IoT cloud services with use of X.509 based mutual authentication. The application example demonstrates ECDSA sign and verify functions using ECC private key stored in ATECC608A

Application Example

Secure Element ECDSA Sign/Verify example: peripherals/secure_element/atecc608_ecdsa.

How to configure and provision ESP32-WROOM-32SE for TLS

To configure and provision ATECC608A chip on ESP32-WROOM-32SE please visit esp_cryptoauth_utility

How to use ATECC608A of ESP32-WROOM-32SE for TLS

ATECC608A can be used for TLS connections using ESP-TLS. To configure ESP-TLS for using secure element please refer *ATECC608A with ESP-TLS* in *ESP-TLS documentation.*

2.6.23 Touch Sensor

Introduction

A touch sensor system is built on a substrate which carries electrodes and relevant connections under a protective flat surface. When a user touches the surface, the capacitance variation is used to evaluate if the touch was valid.

Touch sensor on ESP32 can support up to 10 capacitive touch pads / GPIOs.

The sensing pads can be arranged in different combinations (e.g., matrix, slider), so that a larger area or more points can be detected. The touch pad sensing process is under the control of a hardware-implemented finite-state machine (FSM) which is initiated by software or a dedicated hardware timer.

For design, operation, and control registers of a touch sensor, see *ESP32 Technical Reference Manual* > *On-Chip Sensors and Analog Signal Processing* [PDF].

In-depth design details of touch sensors and firmware development guidelines for ESP32 are available in Touch Sensor Application Note.

For more information about testing touch sensors in various configurations, please check the Guide for ESP32-Sense-Kit.

Functionality Overview

Description of API is broken down into groups of functions to provide a quick overview of the following features:

- Initialization of touch pad driver
- Configuration of touch pad GPIO pins
- Taking measurements
- Adjusting parameters of measurements
- Filtering measurements
- Touch detection methods
- Setting up interrupts to report touch detection
- Waking up from Sleep mode on interrupt

For detailed description of a particular function, please go to Section *API Reference*. Practical implementation of this API is covered in Section *Application Examples*.

Initialization Before using a touch pad, you need to initialize the touch pad driver by calling the function `touch_pad_init()`. This function sets several `..._DEFAULT` driver parameters listed in *API Reference* under *Macros*. It also removes the information about which pads have been touched before, if any, and disables interrupts.

If the driver is not required anymore, deinitialize it by calling `touch_pad_deinit()`.

Configuration Enabling the touch sensor functionality for a particular GPIO is done with `touch_pad_config()`.

Use the function `touch_pad_set_fsm_mode()` to select if touch pad measurement (operated by FSM) should be started automatically by a hardware timer, or by software. If software mode is selected, use `touch_pad_sw_start()` to start the FSM.

Touch State Measurements The following two functions come in handy to read raw or filtered measurements from the sensor:

- `touch_pad_read_raw_data()`
- `touch_pad_read_filtered()`

They can also be used, for example, to evaluate a particular touch pad design by checking the range of sensor readings when a pad is touched or released. This information can be then used to establish a touch threshold.

Note: Before using `touch_pad_read_filtered()`, you need to initialize and configure the filter by calling specific filter functions described in Section *Filtering of Measurements*.

For the demonstration of how to read the touch pad data, check the application example peripherals/touch_sensor/touch_sensor_v1/touch_pad_read.

Method of Measurements The touch sensor will count the number of charge/discharge cycles over a fixed period of time (specified by `touch_pad_set_measurement_clock_cycles()`). The count result is the raw data that read from `touch_pad_read_raw_data()`. After finishing one measurement, the touch sensor will sleep until the next measurement start, this interval between two measurements can be set by `touch_pad_set_measurement_interval()`.

Note: If the specified clock cycles for measurement is too samll, the result may be inaccurate, but increasing clock cycles will increase the power consumption as well. Additionally, the response of the touch sensor will slow down if the total time of the inverval and measurement is too long.

Optimization of Measurements A touch sensor has several configurable parameters to match the characteristics of a particular touch pad design. For instance, to sense smaller capacity changes, it is possible to narrow down the reference voltage range within which the touch pads are charged / discharged. The high and low reference voltages are set using the function `touch_pad_set_voltage()`.

Besides the ability to discern smaller capacity changes, a positive side effect is reduction of power consumption for low power applications. A likely negative effect is an increase in measurement noise. If the dynamic range of obtained readings is still satisfactory, then further reduction of power consumption might be done by reducing the measurement time with `touch_pad_set_measurement_clock_cycles()`.

The following list summarizes available measurement parameters and corresponding 'set' functions:

- Touch pad charge / discharge parameters:
  - voltage range: `touch_pad_set_voltage()`
  - speed (slope): `touch_pad_set_cnt_mode()`
- Clock cycles of one measurement: `touch_pad_set_measurement_clock_cycles()`

Relationship between the voltage range (high / low reference voltages), speed (slope), and measurement time is shown in the figure below.

The last chart *Output* represents the touch sensor reading, i.e., the count of pulses collected within the measurement time.

All functions are provided in pairs to *set* a specific parameter and to *get* the current parameter's value, e.g., `touch_pad_set_voltage()` and `touch_pad_get_voltage()`.

Filtering of Measurements If measurements are noisy, you can filter them with provided API functions. Before using the filter, please start it by calling `touch_pad_filter_start()`.

The filter type is IIR (infinite impulse response), and it has a configurable period that can be set with the function `touch_pad_set_filter_period()`.

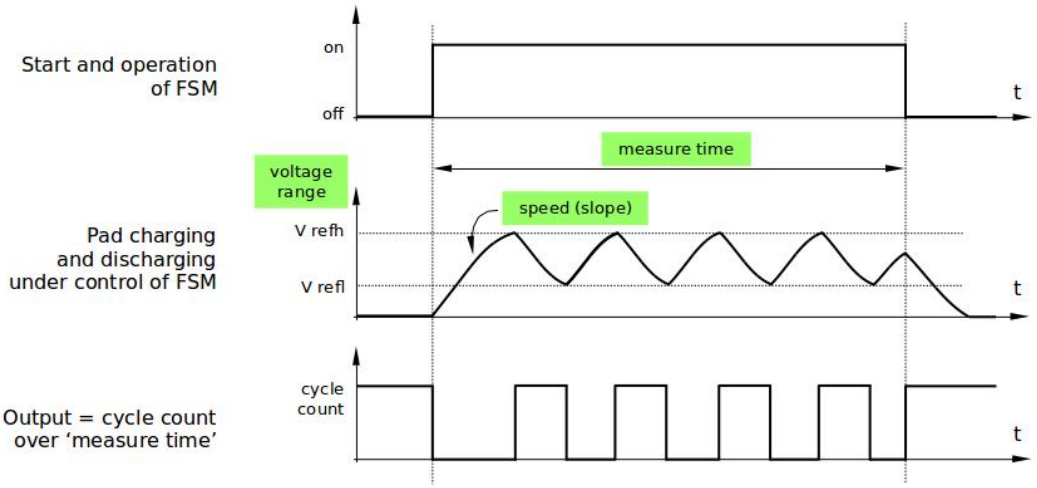


Fig. 22: Touch pad - relationship between measurement parameters

You can stop the filter with `touch_pad_filter_stop()`. If not required anymore, the filter can be deleted by invoking `touch_pad_filter_delete()`.

Touch Detection Touch detection is implemented in ESP32's hardware based on the user-configured threshold and raw measurements executed by FSM. Use the functions `touch_pad_get_status()` to check which pads have been touched and `touch_pad_clear_status()` to clear the touch status information.

Hardware touch detection can also be wired to interrupts. This is described in the next section.

If measurements are noisy and capacity changes are small, hardware touch detection might be unreliable. To resolve this issue, instead of using hardware detection / provided interrupts, implement measurement filtering and perform touch detection in your own application. For sample implementation of both methods of touch detection, see peripherals/touch_sensor/touch_sensor_v1/touch_pad_interrupt.

Touch Triggered Interrupts Before enabling an interrupt on a touch detection, you should establish a touch detection threshold. Use the functions described in *Touch State Measurements* to read and display sensor measurements when a pad is touched and released. Apply a filter if measurements are noisy and relative capacity changes are small. Depending on your application and environment conditions, test the influence of temperature and power supply voltage changes on measured values.

Once a detection threshold is established, it can be set during initialization with `touch_pad_config()` or at the runtime with `touch_pad_set_thresh()`.

In the next step, configure how interrupts are triggered. They can be triggered below or above the threshold, which is set with the function `touch_pad_set_trigger_mode()`.

Finally, configure and manage interrupt calls using the following functions:

- `touch_pad_isr_register()` / `touch_pad_isr_deregister()`
- `touch_pad_intr_enable()` / `touch_pad_intr_disable()`

When interrupts are operational, you can obtain the information from which particular pad an interrupt came by invoking `touch_pad_get_status()` and clear the pad status with `touch_pad_clear_status()`.

Note: Interrupts on touch detection operate on raw / unfiltered measurements checked against user established threshold and are implemented in hardware. Enabling the software filtering API (see *Filtering of Measurements*) does

not affect this process.

Wakeup from Sleep Mode If touch pad interrupts are used to wake up the chip from a sleep mode, you can select a certain configuration of pads (SET1 or both SET1 and SET2) that should be touched to trigger the interrupt and cause the subsequent wakeup. To do so, use the function *touch_pad_set_trigger_source()*.

Configuration of required bit patterns of pads may be managed for each 'SET' with:

- *touch_pad_set_group_mask()* / *touch_pad_get_group_mask()*
- *touch_pad_clear_group_mask()*

Application Examples

- Touch sensor read example: peripherals/touch_sensor/touch_sensor_v1/touch_pad_read.
- Touch sensor interrupt example: peripherals/touch_sensor/touch_sensor_v1/touch_pad_interrupt.

API Reference

Header File

- components/driver/esp32/include/driver/touch_sensor.h

Functions

esp_err_t **touch_pad_config**(*touch_pad_t* touch_num, uint16_t threshold)

Configure touch pad interrupt threshold.

Note: If FSM mode is set to TOUCH_FSM_MODE_TIMER, this function will be blocked for one measurement cycle and wait for data to be valid.

Parameters

- **touch_num** –touch pad index
- **threshold** –interrupt threshold,

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG if argument wrong
- ESP_FAIL if touch pad not initialized

esp_err_t **touch_pad_read**(*touch_pad_t* touch_num, uint16_t *touch_value)

get touch sensor counter value. Each touch sensor has a counter to count the number of charge/discharge cycles. When the pad is not 'touched', we can get a number of the counter. When the pad is 'touched', the value in counter will get smaller because of the larger equivalent capacitance.

Note: This API requests hardware measurement once. If IIR filter mode is enabled, please use 'touch_pad_read_raw_data' interface instead.

Parameters

- **touch_num** –touch pad index
- **touch_value** –pointer to accept touch sensor value

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad parameter error
- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.

- ESP_FAIL Touch pad not initialized

esp_err_t **touch_pad_read_filtered**(*touch_pad_t* touch_num, uint16_t *touch_value)

get filtered touch sensor counter value by IIR filter.

Note: touch_pad_filter_start has to be called before calling touch_pad_read_filtered. This function can be called from ISR

Parameters
- **touch_num** –touch pad index
- **touch_value** –pointer to accept touch sensor value

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad parameter error
- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.
- ESP_FAIL Touch pad not initialized

esp_err_t **touch_pad_read_raw_data**(*touch_pad_t* touch_num, uint16_t *touch_value)

get raw data (touch sensor counter value) from IIR filter process. Need not request hardware measurements.

Note: touch_pad_filter_start has to be called before calling touch_pad_read_raw_data. This function can be called from ISR

Parameters
- **touch_num** –touch pad index
- **touch_value** –pointer to accept touch sensor value

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad parameter error
- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.
- ESP_FAIL Touch pad not initialized

esp_err_t **touch_pad_set_filter_read_cb**(*filter_cb_t* read_cb)

Register the callback function that is called after each IIR filter calculation.

Note: The 'read_cb' callback is called in timer task in each filtering cycle.

Parameters **read_cb** –Pointer to filtered callback function. If the argument passed in is NULL, the callback will stop.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG set error

esp_err_t **touch_pad_isr_register**(*intr_handler_t* fn, void *arg)

Register touch-pad ISR. The handler will be attached to the same CPU core that this function is running on.

Parameters
- **fn** –Pointer to ISR handler
- **arg** –Parameter for ISR

Returns
- ESP_OK Success ;

- ESP_ERR_INVALID_ARG GPIO error
- ESP_ERR_NO_MEM No memory

esp_err_t **touch_pad_set_measurement_clock_cycles**(uint16_t clock_cycle)

Set the clock cycles of each measurement.

Note: This function will specify the clock cycles of each measurement and the clock is sourced from SOC_MOD_CLK_RTC_FAST, its default frequency is SOC_CLK_RC_FAST_FREQ_APPROX The touch sensor will record the charge and discharge times during these clock cycles as the final result (raw value)

Note: If clock cyles is too small, it may lead to inaccurate results.

Parameters **clock_cycle** –The clock cycles of each measurement measure_time = clock_cycle / SOC_CLK_RC_FAST_FREQ_APPROX, the maximum measure time is 0xffff / SOC_CLK_RC_FAST_FREQ_APPROX

Returns

- ESP_OK Set the clock cycle success

esp_err_t **touch_pad_get_measurement_clock_cycles**(uint16_t *clock_cycle)

Get the clock cycles of each measurement.

Parameters **clock_cycle** –The clock cycles of each measurement

Returns

- ESP_OK Get the clock cycle success
- ESP_ERR_INVALID_ARG The input parameter is NULL

esp_err_t **touch_pad_set_measurement_interval**(uint16_t interval_cycle)

Set the interval between two measurements.

Note: The touch sensor will sleep between two mesurements This function is to set the interval cycle And the interval is clocked from SOC_MOD_CLK_RTC_SLOW, its default frequency is SOC_CLK_RC_SLOW_FREQ_APPROX

Parameters **interval_cycle** –The interval between two measurements sleep_time = interval_cycle / SOC_CLK_RC_SLOW_FREQ_APPROX. The approximate frequency value of RTC_SLOW_CLK can be obtained using rtc_clk_slow_freq_get_hz function.

Returns

- ESP_OK Set interval cycle success

esp_err_t **touch_pad_get_measurement_interval**(uint16_t *interval_cycle)

Get the interval between two measurements.

Parameters **interval_cycle** –The interval between two measurements

Returns

- ESP_OK Get interval cycle success
- ESP_ERR_INVALID_ARG The input parameter is NULL

esp_err_t **touch_pad_set_meas_time**(uint16_t sleep_cycle, uint16_t meas_cycle)

Set touch sensor measurement and sleep time. Excessive total time will slow down the touch response. Too small measurement time will not be sampled enough, resulting in inaccurate measurements.

Note: The touch sensor will count the number of charge/discharge cycles over a fixed period of time (specified as the second parameter). That means the number of cycles (raw value) will decrease as the capacity of the touch pad is increasing.

Note: The greater the duty cycle of the measurement time, the more system power is consumed.

Parameters

- **sleep_cycle** –The touch sensor will sleep after each measurement. sleep_cycle decide the interval between each measurement. t_sleep = sleep_cycle / SOC_CLK_RC_SLOW_FREQ_APPROX. The approximate frequency value of RTC_SLOW_CLK can be obtained using rtc_clk_slow_freq_get_hz function.
- **meas_cycle** –The duration of the touch sensor measurement. t_meas = meas_cycle / SOC_CLK_RC_FAST_FREQ_APPROX, the maximum measure time is 0xffff / SOC_CLK_RC_FAST_FREQ_APPROX

Returns

- ESP_OK on success

esp_err_t **touch_pad_get_meas_time**(uint16_t *sleep_cycle, uint16_t *meas_cycle)

Get touch sensor measurement and sleep time.

Parameters

- **sleep_cycle** –Pointer to accept sleep cycle number
- **meas_cycle** –Pointer to accept measurement cycle count.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG The input parameter is NULL

esp_err_t **touch_pad_sw_start**(void)

Trigger a touch sensor measurement, only support in SW mode of FSM.

Returns

- ESP_OK on success

esp_err_t **touch_pad_set_thresh**(*touch_pad_t* touch_num, uint16_t threshold)

Set touch sensor interrupt threshold.

Parameters

- **touch_num** –touch pad index
- **threshold** –threshold of touchpad count, refer to touch_pad_set_trigger_mode to see how to set trigger mode.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_thresh**(*touch_pad_t* touch_num, uint16_t *threshold)

Get touch sensor interrupt threshold.

Parameters

- **touch_num** –touch pad index
- **threshold** –pointer to accept threshold

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_set_trigger_mode**(*touch_trigger_mode_t* mode)

Set touch sensor interrupt trigger mode. Interrupt can be triggered either when counter result is less than threshold or when counter result is more than threshold.

Parameters **mode** –touch sensor interrupt trigger mode

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_trigger_mode**(*touch_trigger_mode_t* *mode)

Get touch sensor interrupt trigger mode.

Parameters **mode** –pointer to accept touch sensor interrupt trigger mode
Returns
- ESP_OK on success

esp_err_t **touch_pad_set_trigger_source**(*touch_trigger_src_t* src)

Set touch sensor interrupt trigger source. There are two sets of touch signals. Set1 and set2 can be mapped to several touch signals. Either set will be triggered if at least one of its touch signal is 'touched'. The interrupt can be configured to be generated if set1 is triggered, or only if both sets are triggered.

Parameters **src** –touch sensor interrupt trigger source
Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_trigger_source**(*touch_trigger_src_t* *src)

Get touch sensor interrupt trigger source.

Parameters **src** –pointer to accept touch sensor interrupt trigger source
Returns
- ESP_OK on success

esp_err_t **touch_pad_set_group_mask**(uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)

Set touch sensor group mask. Touch pad module has two sets of signals, 'Touched' signal is triggered only if at least one of touch pad in this group is "touched". This function will set the register bits according to the given bitmask.

Parameters
- **set1_mask** –bitmask of touch sensor signal group1, it's a 10-bit value
- **set2_mask** –bitmask of touch sensor signal group2, it's a 10-bit value
- **en_mask** –bitmask of touch sensor work enable, it's a 10-bit value

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_group_mask**(uint16_t *set1_mask, uint16_t *set2_mask, uint16_t *en_mask)

Get touch sensor group mask.

Parameters
- **set1_mask** –pointer to accept bitmask of touch sensor signal group1, it's a 10-bit value
- **set2_mask** –pointer to accept bitmask of touch sensor signal group2, it's a 10-bit value
- **en_mask** –pointer to accept bitmask of touch sensor work enable, it's a 10-bit value

Returns
- ESP_OK on success

esp_err_t **touch_pad_clear_group_mask**(uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)

Clear touch sensor group mask. Touch pad module has two sets of signals, Interrupt is triggered only if at least one of touch pad in this group is "touched". This function will clear the register bits according to the given bitmask.

Parameters
- **set1_mask** –bitmask touch sensor signal group1, it's a 10-bit value
- **set2_mask** –bitmask touch sensor signal group2, it's a 10-bit value
- **en_mask** –bitmask of touch sensor work enable, it's a 10-bit value

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_intr_enable**(void)

To enable touch pad interrupt.

Returns
- ESP_OK on success

esp_err_t **touch_pad_intr_disable**(void)

To disable touch pad interrupt.

Returns
- ESP_OK on success

esp_err_t **touch_pad_intr_clear**(void)

To clear touch pad interrupt.

Returns
- ESP_OK on success

esp_err_t **touch_pad_set_filter_period**(uint32_t new_period_ms)

set touch pad filter calibration period, in ms. Need to call touch_pad_filter_start before all touch filter APIs

Parameters **new_period_ms** –filter period, in ms
Returns
- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error
- ESP_ERR_INVALID_ARG parameter error

esp_err_t **touch_pad_get_filter_period**(uint32_t *p_period_ms)

get touch pad filter calibration period, in ms Need to call touch_pad_filter_start before all touch filter APIs

Parameters **p_period_ms** –pointer to accept period
Returns
- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error
- ESP_ERR_INVALID_ARG parameter error

esp_err_t **touch_pad_filter_start**(uint32_t filter_period_ms)

start touch pad filter function This API will start a filter to process the noise in order to prevent false triggering when detecting slight change of capacitance. Need to call touch_pad_filter_start before all touch filter APIs

---

Note: This filter uses FreeRTOS timer, which is dispatched from a task with priority 1 by default on CPU 0. So if some application task with higher priority takes a lot of CPU0 time, then the quality of data obtained from this filter will be affected. You can adjust FreeRTOS timer task priority in menuconfig.

Parameters **filter_period_ms** –filter calibration period, in ms
Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter error
- ESP_ERR_NO_MEM No memory for driver
- ESP_ERR_INVALID_STATE driver state error

esp_err_t **touch_pad_filter_stop**(void)

stop touch pad filter function Need to call touch_pad_filter_start before all touch filter APIs

Returns
- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error

esp_err_t **touch_pad_filter_delete**(void)

delete touch pad filter driver and release the memory Need to call touch_pad_filter_start before all touch filter APIs

Returns
- ESP_OK Success

- ESP_ERR_INVALID_STATE driver state error

Type Definitions

typedef void (***filter_cb_t**)(uint16_t *raw_value, uint16_t *filtered_value)

Callback function that is called after each IIR filter calculation.

Note: This callback is called in timer task in each filtering cycle.

Note: This callback should not be blocked.

Param raw_value The latest raw data(touch sensor counter value) that points to all channels(raw_value[0..TOUCH_PAD_MAX-1]).

Param filtered_value The latest IIR filtered data(calculated from raw data) that points to all channels(filtered_value[0..TOUCH_PAD_MAX-1]).

Header File

- components/driver/include/driver/touch_sensor_common.h

Functions

esp_err_t **touch_pad_init** (void)

Initialize touch module.

Note: If default parameter don't match the usage scenario, it can be changed after this function.

Returns

- ESP_OK Success
- ESP_ERR_NO_MEM Touch pad init error
- ESP_ERR_NOT_SUPPORTED Touch pad is providing current to external XTAL

esp_err_t **touch_pad_deinit** (void)

Un-install touch pad driver.

Note: After this function is called, other touch functions are prohibited from being called.

Returns

- ESP_OK Success
- ESP_FAIL Touch pad driver not initialized

esp_err_t **touch_pad_io_init** (*touch_pad_t* touch_num)

Initialize touch pad GPIO.

Parameters **touch_num** –touch pad index

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_set_voltage**(*touch_high_volt_t* refh, *touch_low_volt_t* refl, *touch_volt_atten_t* atten)

Set touch sensor high voltage threshold of chanrge. The touch sensor measures the channel capacitance value by charging and discharging the channel. So the high threshold should be less than the supply voltage.

Parameters
- **refh** –the value of DREFH
- **refl** –the value of DREFL
- **atten** –the attenuation on DREFH

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_voltage**(*touch_high_volt_t* *refh, *touch_low_volt_t* *refl, *touch_volt_atten_t* *atten)

Get touch sensor reference voltage,.

Parameters
- **refh** –pointer to accept DREFH value
- **refl** –pointer to accept DREFL value
- **atten** –pointer to accept the attenuation on DREFH

Returns
- ESP_OK on success

esp_err_t **touch_pad_set_cnt_mode**(*touch_pad_t* touch_num, *touch_cnt_slope_t* slope, *touch_tie_opt_t* opt)

Set touch sensor charge/discharge speed for each pad. If the slope is 0, the counter would always be zero. If the slope is 1, the charging and discharging would be slow, accordingly. If the slope is set 7, which is the maximum value, the charging and discharging would be fast.

Note: The higher the charge and discharge current, the greater the immunity of the touch channel, but it will increase the system power consumption.

Parameters
- **touch_num** –touch pad index
- **slope** –touch pad charge/discharge speed
- **opt** –the initial voltage

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_cnt_mode**(*touch_pad_t* touch_num, *touch_cnt_slope_t* *slope, *touch_tie_opt_t* *opt)

Get touch sensor charge/discharge speed for each pad.

Parameters
- **touch_num** –touch pad index
- **slope** –pointer to accept touch pad charge/discharge slope
- **opt** –pointer to accept the initial voltage

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_isr_deregister**(void (*fn)(void*), void *arg)

Deregister the handler previously registered using touch_pad_isr_handler_register.

Parameters
- **fn** –handler function to call (as passed to touch_pad_isr_handler_register)
- **arg** –argument of the handler (as passed to touch_pad_isr_handler_register)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if a handler matching both fn and arg isn't registered

esp_err_t **touch_pad_get_wakeup_status**(*touch_pad_t* *pad_num)

Get the touch pad which caused wakeup from deep sleep.

Parameters **pad_num** –pointer to touch pad which caused wakeup

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter is NULL

esp_err_t **touch_pad_set_fsm_mode**(*touch_fsm_mode_t* mode)

Set touch sensor FSM mode, the test action can be triggered by the timer, as well as by the software.

Parameters **mode** –FSM mode

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

esp_err_t **touch_pad_get_fsm_mode**(*touch_fsm_mode_t* *mode)

Get touch sensor FSM mode.

Parameters **mode** –pointer to accept FSM mode

Returns

- ESP_OK on success

esp_err_t **touch_pad_clear_status**(void)

To clear the touch sensor channel active status.

Note: The FSM automatically updates the touch sensor status. It is generally not necessary to call this API to clear the status.

Returns

- ESP_OK on success

uint32_t **touch_pad_get_status**(void)

Get the touch sensor channel active status mask. The bit position represents the channel number. The 0/1 status of the bit represents the trigger status.

Returns

- The touch sensor status. e.g. Touch1 trigger status is `status_mask & (BIT1)`.

bool **touch_pad_meas_is_done**(void)

Check touch sensor measurement status.

Returns

- True measurement is under way
- False measurement done

GPIO Lookup Macros Some useful macros can be used to specified the GPIO number of a touch pad channel, or vice versa. e.g.

1. `TOUCH_PAD_NUM5_GPIO_NUM` is the GPIO number of channel 5 (12);
2. `TOUCH_PAD_GPIO4_CHANNEL` is the channel number of GPIO 4 (channel 0).

Header File

- components/soc/esp32/include/soc/touch_sensor_channel.h

Macros

TOUCH_PAD_GPIO4_CHANNEL

TOUCH_PAD_NUM0_GPIO_NUM

TOUCH_PAD_GPIO0_CHANNEL

TOUCH_PAD_NUM1_GPIO_NUM

TOUCH_PAD_GPIO2_CHANNEL

TOUCH_PAD_NUM2_GPIO_NUM

TOUCH_PAD_GPIO15_CHANNEL

TOUCH_PAD_NUM3_GPIO_NUM

TOUCH_PAD_GPIO13_CHANNEL

TOUCH_PAD_NUM4_GPIO_NUM

TOUCH_PAD_GPIO12_CHANNEL

TOUCH_PAD_NUM5_GPIO_NUM

TOUCH_PAD_GPIO14_CHANNEL

TOUCH_PAD_NUM6_GPIO_NUM

TOUCH_PAD_GPIO27_CHANNEL

TOUCH_PAD_NUM7_GPIO_NUM

TOUCH_PAD_GPIO33_CHANNEL

TOUCH_PAD_NUM8_GPIO_NUM

TOUCH_PAD_GPIO32_CHANNEL

TOUCH_PAD_NUM9_GPIO_NUM

Header File

- components/hal/include/hal/touch_sensor_types.h

Macros

TOUCH_PAD_BIT_MASK_ALL

TOUCH_PAD_SLOPE_DEFAULT

TOUCH_PAD_TIE_OPT_DEFAULT

TOUCH_PAD_BIT_MASK_MAX

TOUCH_PAD_HIGH_VOLTAGE_THRESHOLD

TOUCH_PAD_LOW_VOLTAGE_THRESHOLD

TOUCH_PAD_ATTEN_VOLTAGE_THRESHOLD

TOUCH_PAD_IDLE_CH_CONNECT_DEFAULT

TOUCH_PAD_THRESHOLD_MAX

If set touch threshold max value, The touch sensor can’ t be in touched status

TOUCH_PAD_SLEEP_CYCLE_DEFAULT

The timer frequency is RTC_SLOW_CLK (can be 150k or 32k depending on the options), max value is 0xffff

TOUCH_PAD_MEASURE_CYCLE_DEFAULT

The timer frequency is 8Mhz, the max value is 0x7fff

TOUCH_FSM_MODE_DEFAULT

The touch FSM my be started by the software or timer

TOUCH_TRIGGER_MODE_DEFAULT

Interrupts can be triggered if sensor value gets below or above threshold

TOUCH_TRIGGER_SOURCE_DEFAULT

The wakeup trigger source can be SET1 or both SET1 and SET2

Enumerations

enum **touch_pad_t**

Touch pad channel

Values:

enumerator **TOUCH_PAD_NUM0**

Touch pad channel 0 is GPIO4(ESP32)

enumerator **TOUCH_PAD_NUM1**

Touch pad channel 1 is GPIO0(ESP32) / GPIO1(ESP32-S2)

enumerator **TOUCH_PAD_NUM2**

Touch pad channel 2 is GPIO2(ESP32) / GPIO2(ESP32-S2)

enumerator **TOUCH_PAD_NUM3**

Touch pad channel 3 is GPIO15(ESP32) / GPIO3(ESP32-S2)

enumerator **TOUCH_PAD_NUM4**

Touch pad channel 4 is GPIO13(ESP32) / GPIO4(ESP32-S2)

enumerator **TOUCH_PAD_NUM5**

Touch pad channel 5 is GPIO12(ESP32) / GPIO5(ESP32-S2)

enumerator **TOUCH_PAD_NUM6**

Touch pad channel 6 is GPIO14(ESP32) / GPIO6(ESP32-S2)

enumerator **TOUCH_PAD_NUM7**

Touch pad channel 7 is GPIO27(ESP32) / GPIO7(ESP32-S2)

enumerator **TOUCH_PAD_NUM8**

Touch pad channel 8 is GPIO33(ESP32) / GPIO8(ESP32-S2)

enumerator **TOUCH_PAD_NUM9**

Touch pad channel 9 is GPIO32(ESP32) / GPIO9(ESP32-S2)

enumerator **TOUCH_PAD_MAX**

enum **touch_high_volt_t**

Touch sensor high reference voltage

Values:

enumerator **TOUCH_HVOLT_KEEP**

Touch sensor high reference voltage, no change

enumerator **TOUCH_HVOLT_2V4**

Touch sensor high reference voltage, 2.4V

enumerator **TOUCH_HVOLT_2V5**

Touch sensor high reference voltage, 2.5V

enumerator **TOUCH_HVOLT_2V6**

Touch sensor high reference voltage, 2.6V

enumerator **TOUCH_HVOLT_2V7**

Touch sensor high reference voltage, 2.7V

enumerator **TOUCH_HVOLT_MAX**

enum **touch_low_volt_t**

Touch sensor low reference voltage

Values:

enumerator **TOUCH_LVOLT_KEEP**

Touch sensor low reference voltage, no change

enumerator **TOUCH_LVOLT_0V5**

Touch sensor low reference voltage, 0.5V

enumerator **TOUCH_LVOLT_0V6**

Touch sensor low reference voltage, 0.6V

enumerator **TOUCH_LVOLT_0V7**

Touch sensor low reference voltage, 0.7V

enumerator **TOUCH_LVOLT_0V8**

Touch sensor low reference voltage, 0.8V

enumerator **TOUCH_LVOLT_MAX**

enum **touch_volt_atten_t**

Touch sensor high reference voltage attenuation

Values:

enumerator **TOUCH_HVOLT_ATTEN_KEEP**

Touch sensor high reference voltage attenuation, no change

enumerator **TOUCH_HVOLT_ATTEN_1V5**

Touch sensor high reference voltage attenuation, 1.5V attenuation

enumerator **TOUCH_HVOLT_ATTEN_1V**

Touch sensor high reference voltage attenuation, 1.0V attenuation

enumerator **TOUCH_HVOLT_ATTEN_0V5**

Touch sensor high reference voltage attenuation, 0.5V attenuation

enumerator **TOUCH_HVOLT_ATTEN_0V**

Touch sensor high reference voltage attenuation, 0V attenuation

enumerator **TOUCH_HVOLT_ATTEN_MAX**

enum **touch_cnt_slope_t**

Touch sensor charge/discharge speed

Values:

enumerator **TOUCH_PAD_SLOPE_0**

Touch sensor charge / discharge speed, always zero

enumerator **TOUCH_PAD_SLOPE_1**

Touch sensor charge / discharge speed, slowest

enumerator **TOUCH_PAD_SLOPE_2**

Touch sensor charge / discharge speed

enumerator **TOUCH_PAD_SLOPE_3**

Touch sensor charge / discharge speed

enumerator **TOUCH_PAD_SLOPE_4**

Touch sensor charge / discharge speed

enumerator **TOUCH_PAD_SLOPE_5**

Touch sensor charge / discharge speed

enumerator **TOUCH_PAD_SLOPE_6**

Touch sensor charge / discharge speed

enumerator **TOUCH_PAD_SLOPE_7**

Touch sensor charge / discharge speed, fast

enumerator **TOUCH_PAD_SLOPE_MAX**

enum **touch_tie_opt_t**

Touch sensor initial charge level

Values:

enumerator **TOUCH_PAD_TIE_OPT_LOW**

Initial level of charging voltage, low level

enumerator **TOUCH_PAD_TIE_OPT_HIGH**

Initial level of charging voltage, high level

enumerator **TOUCH_PAD_TIE_OPT_MAX**

enum **touch_fsm_mode_t**

Touch sensor FSM mode

Values:

enumerator **TOUCH_FSM_MODE_TIMER**

To start touch FSM by timer

enumerator **TOUCH_FSM_MODE_SW**

To start touch FSM by software trigger

enumerator **TOUCH_FSM_MODE_MAX**

enum **touch_trigger_mode_t**

Values:

enumerator **TOUCH_TRIGGER_BELOW**

Touch interrupt will happen if counter value is less than threshold.

enumerator **TOUCH_TRIGGER_ABOVE**

Touch interrupt will happen if counter value is larger than threshold.

enumerator **TOUCH_TRIGGER_MAX**

enum **touch_trigger_src_t**

Values:

enumerator **TOUCH_TRIGGER_SOURCE_BOTH**

wakeup interrupt is generated if both SET1 and SET2 are "touched"

enumerator **TOUCH_TRIGGER_SOURCE_SET1**

wakeup interrupt is generated if SET1 is "touched"

enumerator **TOUCH_TRIGGER_SOURCE_MAX**

2.6.24 Two-Wire Automotive Interface (TWAI)

Overview

The Two-Wire Automotive Interface (TWAI) is a real-time serial communication protocol suited for automotive and industrial applications. It is compatible with ISO11898-1 Classical frames, thus can support Standard Frame Format (11-bit ID) and Extended Frame Format (29-bit ID). The ESP32's peripherals contains a TWAI controller that can be configured to communicate on a TWAI bus via an external transceiver.

Warning: The TWAI controller is not compatible with ISO11898-1 FD Format frames, and will interpret such frames as errors.

This programming guide is split into the following sections:

Sections

- *Two-Wire Automotive Interface (TWAI)*
  - *Overview*
  - *TWAI Protocol Summary*
  - *Signals Lines and Transceiver*
  - *Driver Configuration*
  - *Driver Operation*
  - *Examples*
  - *API Reference*

TWAI Protocol Summary

The TWAI is a multi-master, multi-cast, asynchronous, serial communication protocol. TWAI also supports error detection and signalling, and inbuilt message prioritization.

Multi-master: Any node on the bus can initiate the transfer of a message.

Multi-cast: When a node transmits a message, all nodes on the bus will receive the message (i.e., broadcast) thus ensuring data consistency across all nodes. However, some nodes can selectively choose which messages to accept via the use of acceptance filtering (multi-cast).

Asynchronous: The bus does not contain a clock signal. All nodes on the bus operate at the same bit rate and synchronize using the edges of the bits transmitted on the bus.

Error Detection and Signalling: Every node will constantly monitor the bus. When any node detects an error, it will signal the detection by transmitting an error frame. Other nodes will receive the error frame and transmit their own error frames in response. This will result in an error detection being propagated to all nodes on the bus.

Message Priorities: Messages contain an ID field. If two or more nodes attempt to transmit simultaneously, the node transmitting the message with the lower ID value will win arbitration of the bus. All other nodes will become receivers ensuring that there is at most one transmitter at any time.

TWAI Messages TWAI Messages are split into Data Frames and Remote Frames. Data Frames are used to deliver a data payload to other nodes, whereas a Remote Frame is used to request a Data Frame from other nodes (other nodes can optionally respond with a Data Frame). Data and Remote Frames have two frame formats known as **Extended Frame** and **Standard Frame** which contain a 29-bit ID and an 11-bit ID respectively. A TWAI message consists of the following fields:

- 29-bit or 11-bit ID: Determines the priority of the message (lower value has higher priority).
- Data Length Code (DLC) between 0 to 8: Indicates the size (in bytes) of the data payload for a Data Frame, or the amount of data to request for a Remote Frame.
- Up to 8 bytes of data for a Data Frame (should match DLC).

Error States and Counters The TWAI protocol implements a feature known as "fault confinement" where a persistently erroneous node will eventually eliminate itself form the bus. This is implemented by requiring every node to maintain two internal error counters known as the **Transmit Error Counter (TEC)** and the **Receive Error Counter (REC)**. The two error counters are incremented and decremented according to a set of rules (where the counters increase on an error, and decrease on a successful message transmission/reception). The values of the counters are used to determine a node's **error state**, namely **Error Active**, **Error Passive**, and **Bus-Off**.

Error Active: A node is Error Active when **both TEC and REC are less than 128** and indicates that the node is operating normally. Error Active nodes are allowed to participate in bus communications, and will actively signal the detection of any errors by automatically transmitting an **Active Error Flag** over the bus.

Error Passive: A node is Error Passive when **either the TEC or REC becomes greater than or equal to 128**. Error Passive nodes are still able to take part in bus communications, but will instead transmit a **Passive Error Flag** upon detection of an error.

Bus-Off: A node becomes Bus-Off when the **TEC becomes greater than or equal to 256**. A Bus-Off node is unable influence the bus in any manner (essentially disconnected from the bus) thus eliminating itself from the bus. A node will remain in the Bus-Off state until it undergoes bus-off recovery.

Signals Lines and Transceiver

The TWAI controller does not contain a integrated transceiver. Therefore, to connect the TWAI controller to a TWAI bus, **an external transceiver is required**. The type of external transceiver used should depend on the application's physical layer specification (e.g. using SN65HVD23x transceivers for ISO 11898-2 compatibility).

The TWAI controller's interface consists of 4 signal lines known as **TX, RX, BUS-OFF, and CLKOUT**. These four signal lines can be routed through the GPIO Matrix to the ESP32's GPIO pads.

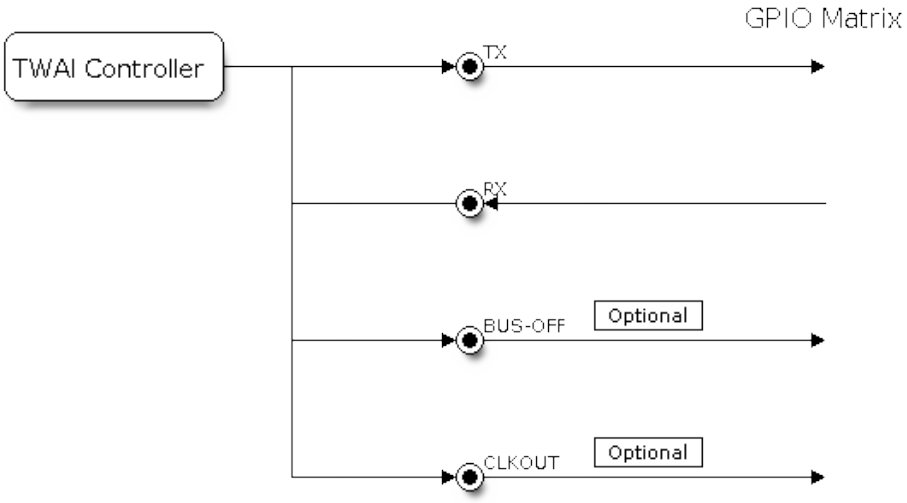


Fig. 23: Signal lines of the TWAI controller

TX and RX: The TX and RX signal lines are required to interface with an external transceiver. Both signal lines represent/interpret a dominant bit as a low logic level (0V), and a recessive bit as a high logic level (3.3V).

BUS-OFF: The BUS-OFF signal line is **optional** and is set to a low logic level (0V) whenever the TWAI controller reaches a bus-off state. The BUS-OFF signal line is set to a high logic level (3.3V) otherwise.

CLKOUT: The CLKOUT signal line is **optional** and outputs a prescaled version of the controller' s source clock (APB Clock).

Note: An external transceiver **must internally loopback the TX to RX** such that a change in logic level to the TX signal line can be observed on the RX line. Failing to do so will cause the TWAI controller to interpret differences in logic levels between the two signal lines as a loss in arbitration or a bit error.

Driver Configuration

This section covers how to configure the TWAI driver.

Operating Modes The TWAI driver supports the following modes of operations:

Normal Mode: The normal operating mode allows the TWAI controller to take part in bus activities such as transmitting and receiving messages/error frames. Acknowledgement from another node is required when transmitting a message.

No Ack Mode: The No Acknowledgement mode is similar to normal mode, however acknowledgements are not required for a message transmission to be considered successful. This mode is useful when self testing the TWAI controller (loopback of transmissions).

Listen Only Mode: This mode will prevent the TWAI controller from influencing the bus. Therefore, transmission of messages/acknowledgement/error frames will be disabled. However the TWAI controller will still be able to receive messages but will not acknowledge the message. This mode is suited for bus monitor applications.

Alerts The TWAI driver contains an alert feature that is used to notify the application layer of certain TWAI controller or TWAI bus events. Alerts are selectively enabled when the TWAI driver is installed, but can be reconfigured

during runtime by calling *twai_reconfigure_alerts()*. The application can then wait for any enabled alerts to occur by calling *twai_read_alerts()*. The TWAI driver supports the following alerts:

Table 7: TWAI Driver Alerts

Alert Flag	Description
TWAI_ALERT_TX_IDLE	No more messages queued for transmission
TWAI_ALERT_TX_SUCCESS	The previous transmission was successful
TWAI_ALERT_RX_DATA	A frame has been received and added to the RX queue
TWAI_ALERT_BELOW_ERR_WARN	Both error counters have dropped below error warning limit
TWAI_ALERT_ERR_ACTIVE	TWAI controller has become error active
TWAI_ALERT_RECOVERY_IN_PROGRESS	TWAI controller is undergoing bus recovery
TWAI_ALERT_BUS_RECOVERED	TWAI controller has successfully completed bus recovery
TWAI_ALERT_ARB_LOST	The previous transmission lost arbitration
TWAI_ALERT_ABOVE_ERR_WARN	One of the error counters have exceeded the error warning limit
TWAI_ALERT_BUS_ERROR	A (Bit, Stuff, CRC, Form, ACK) error has occurred on the bus
TWAI_ALERT_TX_FAILED	The previous transmission has failed
TWAI_ALERT_RX_QUEUE_FULL	The RX queue is full causing a received frame to be lost
TWAI_ALERT_ERR_PASS	TWAI controller has become error passive
TWAI_ALERT_BUS_OFF	Bus-off condition occurred. TWAI controller can no longer influence bus

Note: The TWAI controller's **error warning limit** is used to preemptively warn the application of bus errors before the error passive state is reached. By default, the TWAI driver sets the **error warning limit** to **96**. The TWAI_ALERT_ABOVE_ERR_WARN is raised when the TEC or REC becomes larger then or equal to the error warning limit. The TWAI_ALERT_BELOW_ERR_WARN is raised when both TEC and REC return back to values below **96**.

Note: When enabling alerts, the TWAI_ALERT_AND_LOG flag can be used to cause the TWAI driver to log any raised alerts to UART. However, alert logging is disabled and TWAI_ALERT_AND_LOG if the *CONFIG_TWAI_ISR_IN_IRAM* option is enabled (see *Placing ISR into IRAM*).

Note: The TWAI_ALERT_ALL and TWAI_ALERT_NONE macros can also be used to enable/disable all alerts during configuration/reconfiguration.

Bit Timing The operating bit rate of the TWAI driver is configured using the *twai_timing_config_t* structure. The period of each bit is made up of multiple **time quanta**, and the period of a **time quantum** is determined by a prescaled version of the TWAI controller's source clock. A single bit contains the following segments in the following order:

1. The **Synchronization Segment** consists of a single time quantum
2. **Timing Segment 1** consists of 1 to 16 time quanta before sample point
3. **Timing Segment 2** consists of 1 to 8 time quanta after sample point

The **Baudrate Prescaler** is used to determine the period of each time quantum by dividing the TWAI controller's source clock (80 MHz APB clock). On the ESP32, the brp can be **any even number from 2 to 128**.

If the ESP32 is a revision 2 or later chip, the brp will **also support any multiple of 4 from 132 to 256**, and can be enabled by setting the *CONFIG_ESP32_REV_MIN* to revision 2 or higher.

The sample point of a bit is located on the intersection of Timing Segment 1 and 2. Enabling **Triple Sampling** will cause 3 time quanta to be sampled per bit instead of 1 (extra samples are located at the tail end of Timing Segment 1).

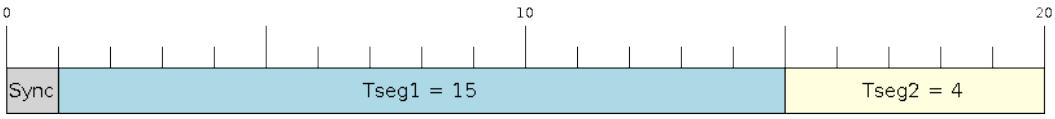


Fig. 24: Bit timing configuration for 500kbit/s given BRP = 8

The **Synchronization Jump Width** is used to determine the maximum number of time quanta a single bit time can be lengthened/shortened for synchronization purposes. `sjw` can **range from 1 to 4**.

Note: Multiple combinations of `brp`, `tseg_1`, `tseg_2`, and `sjw` can achieve the same bit rate. Users should tune these values to the physical characteristics of their bus by taking into account factors such as **propagation delay, node information processing time, and phase errors**.

Bit timing **macro initializers** are also available for commonly used bit rates. The following macro initializers are provided by the TWAI driver.

- `TWAI_TIMING_CONFIG_1MBITS()`
- `TWAI_TIMING_CONFIG_800KBITS()`
- `TWAI_TIMING_CONFIG_500KBITS()`
- `TWAI_TIMING_CONFIG_250KBITS()`
- `TWAI_TIMING_CONFIG_125KBITS()`
- `TWAI_TIMING_CONFIG_100KBITS()`
- `TWAI_TIMING_CONFIG_50KBITS()`
- `TWAI_TIMING_CONFIG_25KBITS()`

Revision 2 or later of the ESP32 also supports the following bit rates:

- `TWAI_TIMING_CONFIG_20KBITS()`
- `TWAI_TIMING_CONFIG_16KBITS()`
- `TWAI_TIMING_CONFIG_12_5KBITS()`

Acceptance Filter The TWAI controller contains a hardware acceptance filter which can be used to filter messages of a particular ID. A node that filters out a message **will not receive the message, but will still acknowledge it**. Acceptance filters can make a node more efficient by filtering out messages sent over the bus that are irrelevant to the node. The acceptance filter is configured using two 32-bit values within *`twai_filter_config_t`* known as the **acceptance code** and the **acceptance mask**.

The **acceptance code** specifies the bit sequence which a message's ID, RTR, and data bytes must match in order for the message to be received by the TWAI controller. The **acceptance mask** is a bit sequence specifying which bits of the acceptance code can be ignored. This allows for a messages of different IDs to be accepted by a single acceptance code.

The acceptance filter can be used under **Single or Dual Filter Mode**. Single Filter Mode will use the acceptance code and mask to define a single filter. This allows for the first two data bytes of a standard frame to be filtered, or the entirety of an extended frame's 29-bit ID. The following diagram illustrates how the 32-bit acceptance code and mask will be interpreted under Single Filter Mode (Note: The yellow and blue fields represent standard and extended frame formats respectively).

Dual Filter Mode will use the acceptance code and mask to define two separate filters allowing for increased flexibility of ID's to accept, but does not allow for all 29-bits of an extended ID to be filtered. The following diagram illustrates how the 32-bit acceptance code and mask will be interpreted under **Dual Filter Mode** (Note: The yellow and blue fields represent standard and extended frame formats respectively).

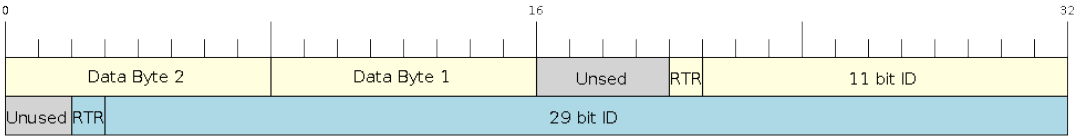


Fig. 25: Bit layout of single filter mode (Right side MSBit)

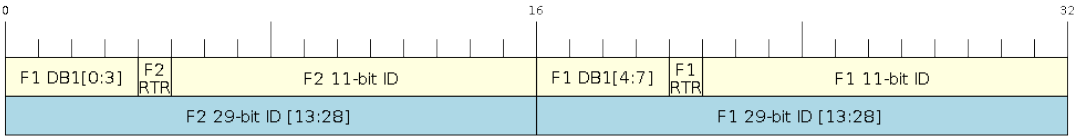


Fig. 26: Bit layout of dual filter mode (Right side MSBit)

Disabling TX Queue The TX queue can be disabled during configuration by setting the `tx_queue_len` member of *`twai_general_config_t`* to `0`. This will allow applications that do not require message transmission to save a small amount of memory when using the TWAI driver.

Placing ISR into IRAM The TWAI driver's ISR (Interrupt Service Routine) can be placed into IRAM so that the ISR can still run whilst the cache is disabled. Placing the ISR into IRAM may be necessary to maintain the TWAI driver's functionality during lengthy cache disabling operations (such as SPI Flash writes, OTA updates etc). Whilst the cache is disabled, the ISR will continue to:

- Read received messages from the RX buffer and place them into the driver's RX queue.
- Load messages pending transmission from the driver's TX queue and write them into the TX buffer.

To place the TWAI driver's ISR, users must do the following:

- Enable the *CONFIG_TWAI_ISR_IN_IRAM* option using `idf.py menuconfig`.
- When calling *`twai_driver_install()`*, the *intr_flags* member of *`twai_general_config_t`* should set the *`ESP_INTR_FLAG_IRAM`* set.

Note: When the *CONFIG_TWAI_ISR_IN_IRAM* option is enabled, the TWAI driver will no longer log any alerts (i.e., the `TWAI_ALERT_AND_LOG` flag will not have any effect).

ESP32 Errata Workarounds The ESP32's TWAI controller contains multiple hardware errata (more details about the errata can be found in the ESP32's ECO document). Some of these errata are critical, and under specific circumstances, can place the TWAI controller into an unrecoverable state (i.e., the controller gets stuck until it is reset by the CPU).

The TWAI driver contains software workarounds for these critical errata. With these workarounds, the ESP32 TWAI driver can operate normally, albeit with degraded performance. The degraded performance will affect users in the following ways depending on what particular errata conditions are encountered:

- The TWAI driver can occasionally drop some received messages.
- The TWAI driver can be unresponsive for a short period of time (i.e., will not transmit or ACK for 11 bit times or longer).
- If *CONFIG_TWAI_ISR_IN_IRAM* is enabled, the workarounds will increase IRAM usage by approximately 1KB.

The software workarounds are enabled by default and it is recommended that users keep this workarounds enabled.

Driver Operation

The TWAI driver is designed with distinct states and strict rules regarding the functions or conditions that trigger a state transition. The following diagram illustrates the various states and their transitions.

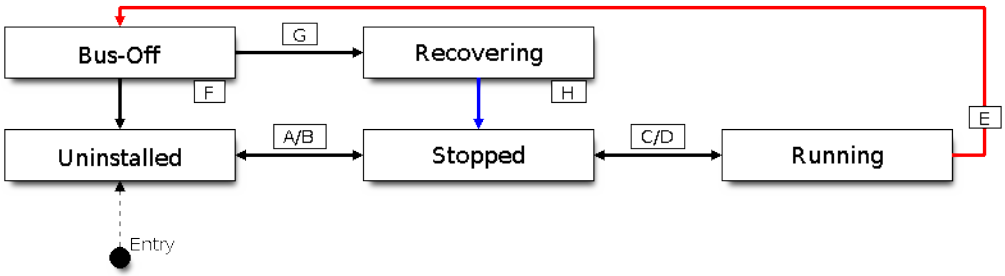


Fig. 27: State transition diagram of the TWAI driver (see table below)

Label	Transition	Action/Condition
A	Uninstalled -> Stopped	`twai_driver_install()`
B	Stopped -> Uninstalled	`twai_driver_uninstall()`
C	Stopped -> Running	`twai_start()`
D	Running -> Stopped	`twai_stop()`
E	Running -> Bus-Off	Transmit Error Counter >= 256
F	Bus-Off -> Uninstalled	`twai_driver_uninstall()`
G	Bus-Off -> Recovering	`twai_initiate_recovery()`
H	Recovering -> Stopped	128 occurrences of 11 consecutive recessive bits.

Driver States **Uninstalled**: In the uninstalled state, no memory is allocated for the driver and the TWAI controller is powered OFF.

Stopped: In this state, the TWAI controller is powered ON and the TWAI driver has been installed. However the TWAI controller will be unable to take part in any bus activities such as transmitting, receiving, or acknowledging messages.

Running: In the running state, the TWAI controller is able to take part in bus activities. Therefore messages can be transmitted/received/acknowledged. Furthermore the TWAI controller will be able to transmit error frames upon detection of errors on the bus.

Bus-Off: The bus-off state is automatically entered when the TWAI controller' s Transmit Error Counter becomes greater than or equal to 256. The bus-off state indicates the occurrence of severe errors on the bus or in the TWAI controller. Whilst in the bus-off state, the TWAI controller will be unable to take part in any bus activities. To exit the bus-off state, the TWAI controller must undergo the bus recovery process.

Recovering: The recovering state is entered when the TWAI controller undergoes bus recovery. The TWAI controller/TWAI driver will remain in the recovering state until the 128 occurrences of 11 consecutive recessive bits is observed on the bus.

Message Fields and Flags The TWAI driver distinguishes different types of messages by using the various bit field members of the `twai_message_t` structure. These bit field members determine whether a message is in standard or extended format, a remote frame, and the type of transmission to use when transmitting such a message.

These bit field members can also be toggled using the the *flags* member of `twai_message_t` and the following message flags:

Message Flag	Description
TWAI_MSG_FLAG_EXTD	Message is in Extended Frame Format (29bit ID)
TWAI_MSG_FLAG_RTR	Message is a Remote Frame (Remote Transmission Request)
TWAI_MSG_FLAG_SS	Transmit message using Single Shot Transmission (Message will not be re-transmitted upon error or loss of arbitration). Unused for received message.
TWAI_MSG_FLAG_SELF	Transmit message using Self Reception Request (Transmitted message will also received by the same node). Unused for received message.
TWAI_MSG_FLAG_DLC_NON_COMP	Message's Data length code is larger than 8. This will break compliance with TWAI
TWAI_MSG_FLAG_NONE	Clears all bit fields. Equivalent to a Standard Frame Format (11bit ID) Data Frame.

Examples

Configuration & Installation The following code snippet demonstrates how to configure, install, and start the TWAI driver via the use of the various configuration structures, macro initializers, the *twai_driver_install()* function, and the *twai_start()* function.

```
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    //Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_21, GPIO_
↪NUM_22, TWAI_MODE_NORMAL);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    //Install TWAI driver
    if (twai_driver_install(&g_config, &t_config, &f_config) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    //Start TWAI driver
    if (twai_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    ...

}
```

The usage of macro initializers is not mandatory and each of the configuration structures can be manually.

Message Transmission The following code snippet demonstrates how to transmit a message via the usage of the *twai_message_t* type and *twai_transmit()* function.

```
#include "driver/twai.h"

...
```

(continues on next page)

(continued from previous page)

```
//Configure message to transmit
twai_message_t message;
message.identifier = 0xAAAA;
message.extd = 1;
message.data_length_code = 4;
for (int i = 0; i < 4; i++) {
    message.data[i] = 0;
}

//Queue message for transmission
if (twai_transmit(&message, pdMS_TO_TICKS(1000)) == ESP_OK) {
    printf("Message queued for transmission\n");
} else {
    printf("Failed to queue message for transmission\n");
}
```

Message Reception The following code snippet demonstrates how to receive a message via the usage of the *twai_message_t* type and *twai_receive()* function.

```
#include "driver/twai.h"

...

//Wait for message to be received
twai_message_t message;
if (twai_receive(&message, pdMS_TO_TICKS(10000)) == ESP_OK) {
    printf("Message received\n");
} else {
    printf("Failed to receive message\n");
    return;
}

//Process received message
if (message.extd) {
    printf("Message is in Extended Format\n");
} else {
    printf("Message is in Standard Format\n");
}
printf("ID is %d\n", message.identifier);
if (!(message.rtr)) {
    for (int i = 0; i < message.data_length_code; i++) {
        printf("Data byte %d = %d\n", i, message.data[i]);
    }
}
```

Reconfiguring and Reading Alerts The following code snippet demonstrates how to reconfigure and read TWAI driver alerts via the use of the *twai_reconfigure_alerts()* and *twai_read_alerts()* functions.

```
#include "driver/twai.h"

...

//Reconfigure alerts to detect Error Passive and Bus-Off error states
uint32_t alerts_to_enable = TWAI_ALERT_ERR_PASS | TWAI_ALERT_BUS_OFF;
if (twai_reconfigure_alerts(alerts_to_enable, NULL) == ESP_OK) {
    printf("Alerts reconfigured\n");
} else {
    printf("Failed to reconfigure alerts");
```

(continues on next page)

(continued from previous page)

```
}

//Block indefinitely until an alert occurs
uint32_t alerts_triggered;
twai_read_alerts(&alerts_triggered, portMAX_DELAY);
```

Stop and Uninstall The following code demonstrates how to stop and uninstall the TWAI driver via the use of the *twai_stop()* and *twai_driver_uninstall()* functions.

```
#include "driver/twai.h"

...

//Stop the TWAI driver
if (twai_stop() == ESP_OK) {
    printf("Driver stopped\n");
} else {
    printf("Failed to stop driver\n");
    return;
}

//Uninstall the TWAI driver
if (twai_driver_uninstall() == ESP_OK) {
    printf("Driver uninstalled\n");
} else {
    printf("Failed to uninstall driver\n");
    return;
}
```

Multiple ID Filter Configuration The acceptance mask in *twai_filter_config_t* can be configured such that two or more IDs will be accepted for a single filter. For a particular filter to accept multiple IDs, the conflicting bit positions amongst the IDs must be set in the acceptance mask. The acceptance code can be set to any one of the IDs.

The following example shows how the calculate the acceptance mask given multiple IDs:

```
ID1 =  11'b101 1010 0000
ID2 =  11'b101 1010 0001
ID3 =  11'b101 1010 0100
ID4 =  11'b101 1010 1000
//Acceptance Mask
MASK = 11'b000 0000 1101
```

Application Examples **Network Example:** The TWAI Network example demonstrates communication between two ESP32s using the TWAI driver API. One TWAI node acts as a network master that initiates and ceases the transfer of a data from another node acting as a network slave. The example can be found via peripherals/twai/twai_network.

Alert and Recovery Example: This example demonstrates how to use the TWAI driver's alert and bus-off recovery API. The example purposely introduces errors on the bus to put the TWAI controller into the Bus-Off state. An alert is used to detect the Bus-Off state and trigger the bus recovery process. The example can be found via peripherals/twai/twai_alert_and_recovery.

Self Test Example: This example uses the No Acknowledge Mode and Self Reception Request to cause the TWAI controller to send and simultaneously receive a series of messages. This example can be used to verify if the connections between the TWAI controller and the external transceiver are working correctly. The example can be found via peripherals/twai/twai_self_test.

API Reference

Header File

- components/hal/include/hal/twai_types.h

Structures

struct **twai_message_t**

Structure to store a TWAI message.

Note: The flags member is deprecated

Public Members

uint32_t **extd**

Extended Frame Format (29bit ID)

uint32_t **rtr**

Message is a Remote Frame

uint32_t **ss**

Transmit as a Single Shot Transmission. Unused for received.

uint32_t **self**

Transmit as a Self Reception Request. Unused for received.

uint32_t **dlc_non_comp**

Message's Data length code is larger than 8. This will break compliance with ISO 11898-1

uint32_t **reserved**

Reserved bits

uint32_t **flags**

Deprecated: Alternate way to set bits using message flags

uint32_t **identifier**

11 or 29 bit identifier

uint8_t **data_length_code**

Data length code

uint8_t **data**[TWAI_FRAME_MAX_DLC]

Data bytes (not relevant in RTR frame)

struct **twai_timing_config_t**

Structure for bit timing configuration of the TWAI driver.

Note: Macro initializers are available for this structure

Public Members

uint32_t **brp**

Baudrate prescaler (i.e., APB clock divider). Any even number from 2 to 128 for ESP32, 2 to 32768 for ESP32S2. For ESP32 Rev 2 or later, multiples of 4 from 132 to 256 are also supported

uint8_t **tseg_1**

Timing segment 1 (Number of time quanta, between 1 to 16)

uint8_t **tseg_2**

Timing segment 2 (Number of time quanta, 1 to 8)

uint8_t **sjw**

Synchronization Jump Width (Max time quanta jump for synchronize from 1 to 4)

bool **triple_sampling**

Enables triple sampling when the TWAI controller samples a bit

struct **twai_filter_config_t**

Structure for acceptance filter configuration of the TWAI driver (see documentation)

Note: Macro initializers are available for this structure

Public Members

uint32_t **acceptance_code**

32-bit acceptance code

uint32_t **acceptance_mask**

32-bit acceptance mask

bool **single_filter**

Use Single Filter Mode (see documentation)

Macros

TWAI_EXTD_ID_MASK

TWAI Constants.

Bit mask for 29 bit Extended Frame Format ID

TWAI_STD_ID_MASK

Bit mask for 11 bit Standard Frame Format ID

TWAI_FRAME_MAX_DLC

Max data bytes allowed in TWAI

TWAI_FRAME_EXTD_ID_LEN_BYTES

EFF ID requires 4 bytes (29bit)

TWAI_FRAME_STD_ID_LEN_BYTES

SFF ID requires 2 bytes (11bit)

TWAI_ERR_PASS_THRESH

Error counter threshold for error passive

Enumerations

enum **twai_mode_t**

TWAI Controller operating modes.

Values:

enumerator **TWAI_MODE_NORMAL**

Normal operating mode where TWAI controller can send/receive/acknowledge messages

enumerator **TWAI_MODE_NO_ACK**

Transmission does not require acknowledgment. Use this mode for self testing

enumerator **TWAI_MODE_LISTEN_ONLY**

The TWAI controller will not influence the bus (No transmissions or acknowledgments) but can receive messages

Header File

- components/driver/include/driver/twai.h

Functions

esp_err_t **twai_driver_install** (const *twai_general_config_t* *g_config, const *twai_timing_config_t* *t_config, const *twai_filter_config_t* *f_config)

Install TWAI driver.

This function installs the TWAI driver using three configuration structures. The required memory is allocated and the TWAI driver is placed in the stopped state after running this function.

Note: Macro initializers are available for the configuration structures (see documentation)

Note: To reinstall the TWAI driver, call twai_driver_uninstall() first

Parameters

- **g_config** –**[in]** General configuration structure
- **t_config** –**[in]** Timing configuration structure
- **f_config** –**[in]** Filter configuration structure

Returns

- ESP_OK: Successfully installed TWAI driver
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_STATE: Driver is already installed

esp_err_t **twai_driver_uninstall** (void)

Uninstall the TWAI driver.

This function uninstalls the TWAI driver, freeing the memory utilized by the driver. This function can only be called when the driver is in the stopped state or the bus-off state.

Warning: The application must ensure that no tasks are blocked on TX/RX queues or alerts when this function is called.

Returns

- ESP_OK: Successfully uninstalled TWAI driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

esp_err_t **twai_start** (void)

Start the TWAI driver.

This function starts the TWAI driver, putting the TWAI driver into the running state. This allows the TWAI driver to participate in TWAI bus activities such as transmitting/receiving messages. The TX and RX queue are reset in this function, clearing any messages that are unread or pending transmission. This function can only be called when the TWAI driver is in the stopped state.

Returns

- ESP_OK: TWAI driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

esp_err_t **twai_stop** (void)

Stop the TWAI driver.

This function stops the TWAI driver, preventing any further message from being transmitted or received until twai_start() is called. Any messages in the TX queue are cleared. Any messages in the RX queue should be read by the application after this function is called. This function can only be called when the TWAI driver is in the running state.

Warning: A message currently being transmitted/received on the TWAI bus will be ceased immediately. This may lead to other TWAI nodes interpreting the unfinished message as an error.

Returns

- ESP_OK: TWAI driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

esp_err_t **twai_transmit** (const *twai_message_t* *message, TickType_t ticks_to_wait)

Transmit a TWAI message.

This function queues a TWAI message for transmission. Transmission will start immediately if no other messages are queued for transmission. If the TX queue is full, this function will block until more space becomes available or until it times out. If the TX queue is disabled (TX queue length = 0 in configuration), this function will return immediately if another message is undergoing transmission. This function can only be called when the TWAI driver is in the running state and cannot be called under Listen Only Mode.

Note: This function does not guarantee that the transmission is successful. The TX_SUCCESS/TX_FAILED alert can be enabled to alert the application upon the success/failure of a transmission.

Note: The TX_IDLE alert can be used to alert the application when no other messages are awaiting transmission.

Parameters

- **message** –**[in]** Message to transmit
- **ticks_to_wait** –**[in]** Number of FreeRTOS ticks to block on the TX queue

Returns

- ESP_OK: Transmission successfully queued/initiated
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_TIMEOUT: Timed out waiting for space on TX queue
- ESP_FAIL: TX queue is disabled and another message is currently transmitting
- ESP_ERR_INVALID_STATE: TWAI driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

esp_err_t **twai_receive**(*twai_message_t* *message, TickType_t ticks_to_wait)

Receive a TWAI message.

This function receives a message from the RX queue. The flags field of the message structure will indicate the type of message received. This function will block if there are no messages in the RX queue

Warning: The flags field of the received message should be checked to determine if the received message contains any data bytes.

Parameters

- **message** –**[out]** Received message
- **ticks_to_wait** –**[in]** Number of FreeRTOS ticks to block on RX queue

Returns

- ESP_OK: Message successfully received from RX queue
- ESP_ERR_TIMEOUT: Timed out waiting for message
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

esp_err_t **twai_read_alerts**(uint32_t *alerts, TickType_t ticks_to_wait)

Read TWAI driver alerts.

This function will read the alerts raised by the TWAI driver. If no alert has been issued when this function is called, this function will block until an alert occurs or until it timeouts.

Note: Multiple alerts can be raised simultaneously. The application should check for all alerts that have been enabled.

Parameters

- **alerts** –**[out]** Bit field of raised alerts (see documentation for alert flags)
- **ticks_to_wait** –**[in]** Number of FreeRTOS ticks to block for alert

Returns

- ESP_OK: Alerts read
- ESP_ERR_TIMEOUT: Timed out waiting for alerts
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

esp_err_t **twai_reconfigure_alerts**(uint32_t alerts_enabled, uint32_t *current_alerts)

Reconfigure which alerts are enabled.

This function reconfigures which alerts are enabled. If there are alerts which have not been read whilst reconfiguring, this function can read those alerts.

Parameters

- **alerts_enabled** –**[in]** Bit field of alerts to enable (see documentation for alert flags)
- **current_alerts** –**[out]** Bit field of currently raised alerts. Set to NULL if unused

Returns
- ESP_OK: Alerts reconfigured
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

esp_err_t **twai_initiate_recovery**(void)

Start the bus recovery process.

This function initiates the bus recovery process when the TWAI driver is in the bus-off state. Once initiated, the TWAI driver will enter the recovering state and wait for 128 occurrences of the bus-free signal on the TWAI bus before returning to the stopped state. This function will reset the TX queue, clearing any messages pending transmission.

Note: The BUS_RECOVERED alert can be enabled to alert the application when the bus recovery process completes.

Returns
- ESP_OK: Bus recovery started
- ESP_ERR_INVALID_STATE: TWAI driver is not in the bus-off state, or is not installed

esp_err_t **twai_get_status_info**(*twai_status_info_t* *status_info)

Get current status information of the TWAI driver.

Parameters **status_info** – **[out]** Status information

Returns
- ESP_OK: Status information retrieved
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

esp_err_t **twai_clear_transmit_queue**(void)

Clear the transmit queue.

This function will clear the transmit queue of all messages.

Note: The transmit queue is automatically cleared when twai_stop() or twai_initiate_recovery() is called.

Returns
- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed or TX queue is disabled

esp_err_t **twai_clear_receive_queue**(void)

Clear the receive queue.

This function will clear the receive queue of all messages.

Note: The receive queue is automatically cleared when twai_start() is called.

Returns
- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

Structures

struct **twai_general_config_t**

Structure for general configuration of the TWAI driver.

Note: Macro initializers are available for this structure

Public Members

twai_mode_t **mode**

Mode of TWAI controller

gpio_num_t **tx_io**

Transmit GPIO number

gpio_num_t **rx_io**

Receive GPIO number

gpio_num_t **clkout_io**

CLKOUT GPIO number (optional, set to -1 if unused)

gpio_num_t **bus_off_io**

Bus off indicator GPIO number (optional, set to -1 if unused)

uint32_t **tx_queue_len**

Number of messages TX queue can hold (set to 0 to disable TX Queue)

uint32_t **rx_queue_len**

Number of messages RX queue can hold

uint32_t **alerts_enabled**

Bit field of alerts to enable (see documentation)

uint32_t **clkout_divider**

CLKOUT divider. Can be 1 or any even number from 2 to 14 (optional, set to 0 if unused)

int **intr_flags**

Interrupt flags to set the priority of the driver's ISR. Note that to use the ESP_INTR_FLAG_IRAM, the CONFIG_TWAI_ISR_IN_IRAM option should be enabled first.

struct **twai_status_info_t**

Structure to store status information of TWAI driver.

Public Members

twai_state_t **state**

Current state of TWAI controller (Stopped/Running/Bus-Off/Recovery)

uint32_t **msgs_to_tx**

Number of messages queued for transmission or awaiting transmission completion

uint32_t **msgs_to_rx**

Number of messages in RX queue waiting to be read

uint32_t **tx_error_counter**

Current value of Transmit Error Counter

uint32_t **rx_error_counter**

Current value of Receive Error Counter

uint32_t **tx_failed_count**

Number of messages that failed transmissions

uint32_t **rx_missed_count**

Number of messages that were lost due to a full RX queue (or errata workaround if enabled)

uint32_t **rx_overrun_count**

Number of messages that were lost due to a RX FIFO overrun

uint32_t **arb_lost_count**

Number of instances arbitration was lost

uint32_t **bus_error_count**

Number of instances a bus error has occurred

Macros

TWAI_IO_UNUSED

Marks GPIO as unused in TWAI configuration

Enumerations

enum **twai_state_t**

TWAI driver states.

Values:

enumerator **TWAI_STATE_STOPPED**

Stopped state. The TWAI controller will not participate in any TWAI bus activities

enumerator **TWAI_STATE_RUNNING**

Running state. The TWAI controller can transmit and receive messages

enumerator **TWAI_STATE_BUS_OFF**

Bus-off state. The TWAI controller cannot participate in bus activities until it has recovered

enumerator **TWAI_STATE_RECOVERING**

Recovering state. The TWAI controller is undergoing bus recovery

2.6.25 Universal Asynchronous Receiver/Transmitter (UART)

Overview

A Universal Asynchronous Receiver/Transmitter (UART) is a hardware feature that handles communication (i.e., timing requirements and data framing) using widely-adopted asynchronous serial communication interfaces, such as RS232, RS422, RS485. A UART provides a widely adopted and cheap method to realize full-duplex or half-duplex data exchange among different devices.

The ESP32 chip has three UART controllers (also referred to as port), each featuring an identical set of registers to simplify programming and for more flexibility.

Each UART controller is independently configurable with parameters such as baud rate, data bit length, bit ordering, number of stop bits, parity bit etc. All the controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association protocols (IrDA).

Functional Overview

The following overview describes how to establish communication between an ESP32 and other UART devices using the functions and data types of the UART driver. The overview reflects a typical programming workflow and is broken down into the sections provided below:

1. *Setting Communication Parameters* - Setting baud rate, data bits, stop bits, etc.
2. *Setting Communication Pins* - Assigning pins for connection to a device.
3. *Driver Installation* - Allocating ESP32's resources for the UART driver.
4. *Running UART Communication* - Sending / receiving data
5. *Using Interrupts* - Triggering interrupts on specific communication events
6. *Deleting a Driver* - Freeing allocated resources if a UART communication is no longer required

Steps 1 to 3 comprise the configuration stage. Step 4 is where the UART starts operating. Steps 5 and 6 are optional.

The UART driver's functions identify each of the UART controllers using `uart_port_t`. This identification is needed for all the following function calls.

Setting Communication Parameters UART communication parameters can be configured all in a single step or individually in multiple steps.

Single Step Call the function `uart_param_config()` and pass to it a `uart_config_t` structure. The `uart_config_t` structure should contain all the required parameters. See the example below.

```
const uart_port_t uart_num = UART_NUM_2;
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
    .rx_flow_ctrl_thresh = 122,
};
// Configure UART parameters
ESP_ERROR_CHECK(uart_param_config(uart_num, &uart_config));
```

For more information on how to configure the hardware flow control options, please refer to peripherals/uart/uart_echo.

Multiple Steps Configure specific parameters individually by calling a dedicated function from the table given below. These functions are also useful if re-configuring a single parameter.

Table 8: Functions for Configuring specific parameters individually

Parameter to Configure	Function
Baud rate	*uart_set_baudrate()*
Number of transmitted bits	*uart_set_word_length()* selected out of *uart_word_length_t*
Parity control	*uart_set_parity()* selected out of *uart_parity_t*
Number of stop bits	*uart_set_stop_bits()* selected out of *uart_stop_bits_t*
Hardware flow control mode	*uart_set_hw_flow_ctrl()* selected out of *uart_hw_flowcontrol_t*
Communication mode	*uart_set_mode()* selected out of *uart_mode_t*

Each of the above functions has a _get_ counterpart to check the currently set value. For example, to check the current baud rate value, call *uart_get_baudrate()*.

Setting Communication Pins After setting communication parameters, configure the physical GPIO pins to which the other UART device will be connected. For this, call the function *uart_set_pin()* and specify the GPIO pin numbers to which the driver should route the Tx, Rx, RTS, and CTS signals. If you want to keep a currently allocated pin number for a specific signal, pass the macro *UART_PIN_NO_CHANGE*.

The same macro should be specified for pins that will not be used.

```
// Set UART pins(TX: IO4, RX: IO5, RTS: IO18, CTS: IO19)
ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2, 4, 5, 18, 19));
```

Driver Installation Once the communication pins are set, install the driver by calling *uart_driver_install()* and specify the following parameters:

- Size of Tx ring buffer
- Size of Rx ring buffer
- Event queue handle and size
- Flags to allocate an interrupt

The function will allocate the required internal resources for the UART driver.

```
// Setup UART buffered IO with event queue
const int uart_buffer_size = (1024 * 2);
QueueHandle_t uart_queue;
// Install UART driver using an event queue here
ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2, uart_buffer_size, \
                                    uart_buffer_size, 10, &uart_queue, 0));
```

Once this step is complete, you can connect the external UART device and check the communication.

Running UART Communication Serial communication is controlled by each UART controller's finite state machine (FSM).

The process of sending data involves the following steps:

1. Write data into Tx FIFO buffer
2. FSM serializes the data
3. FSM sends the data out

The process of receiving data is similar, but the steps are reversed:

1. FSM processes an incoming serial stream and parallelizes it
2. FSM writes the data into Rx FIFO buffer
3. Read the data from Rx FIFO buffer

Therefore, an application will be limited to writing and reading data from a respective buffer using *uart_write_bytes()* and *uart_read_bytes()* respectively, and the FSM will do the rest.

Transmitting After preparing the data for transmission, call the function *uart_write_bytes()* and pass the data buffer's address and data length to it. The function will copy the data to the Tx ring buffer (either immediately or after enough space is available), and then exit. When there is free space in the Tx FIFO buffer, an interrupt service routine (ISR) moves the data from the Tx ring buffer to the Tx FIFO buffer in the background. The code below demonstrates the use of this function.

```
// Write data to UART.
char* test_str = "This is a test string.\n";
uart_write_bytes(uart_num, (const char*)test_str, strlen(test_str));
```

The function *uart_write_bytes_with_break()* is similar to *uart_write_bytes()* but adds a serial break signal at the end of the transmission. A 'serial break signal' means holding the Tx line low for a period longer than one data frame.

```
// Write data to UART, end with a break signal.
uart_write_bytes_with_break(uart_num, "test break\n",strlen("test break\n"), 100);
```

Another function for writing data to the Tx FIFO buffer is *uart_tx_chars()*. Unlike *uart_write_bytes()*, this function will not block until space is available. Instead, it will write all data which can immediately fit into the hardware Tx FIFO, and then return the number of bytes that were written.

There is a 'companion' function *uart_wait_tx_done()* that monitors the status of the Tx FIFO buffer and returns once it is empty.

```
// Wait for packet to be sent
const uart_port_t uart_num = UART_NUM_2;
ESP_ERROR_CHECK(uart_wait_tx_done(uart_num, 100)); // wait timeout is 100 RTOS␣
↪ticks (TickType_t)
```

Receiving Once the data is received by the UART and saved in the Rx FIFO buffer, it needs to be retrieved using the function *uart_read_bytes()*. Before reading data, you can check the number of bytes available in the Rx FIFO buffer by calling *uart_get_buffered_data_len()*. An example of using these functions is given below.

```
// Read data from UART.
const uart_port_t uart_num = UART_NUM_2;
uint8_t data[128];
int length = 0;
ESP_ERROR_CHECK(uart_get_buffered_data_len(uart_num, (size_t*)&length));
length = uart_read_bytes(uart_num, data, length, 100);
```

If the data in the Rx FIFO buffer is no longer needed, you can clear the buffer by calling *uart_flush()*.

Software Flow Control If the hardware flow control is disabled, you can manually set the RTS and DTR signal levels by using the functions *uart_set_rts()* and *uart_set_dtr()* respectively.

Communication Mode Selection The UART controller supports a number of communication modes. A mode can be selected using the function *uart_set_mode()*. Once a specific mode is selected, the UART driver will handle the behavior of a connected UART device accordingly. As an example, it can control the RS485 driver chip using the RTS line to allow half-duplex RS485 communication.

```
// Setup UART in rs485 half duplex mode
ESP_ERROR_CHECK(uart_set_mode(uart_num, UART_MODE_RS485_HALF_DUPLEX));
```

Using Interrupts There are many interrupts that can be generated following specific UART states or detected errors. The full list of available interrupts is provided in *ESP32 Technical Reference Manual* > *UART Controller*

(UART) > UART Interrupts and *UHCI Interrupts* [PDF]. You can enable or disable specific interrupts by calling `uart_enable_intr_mask()` or `uart_disable_intr_mask()` respectively.

The `uart_driver_install()` function installs the driver's internal interrupt handler to manage the Tx and Rx ring buffers and provides high-level API functions like events (see below).

The API provides a convenient way to handle specific interrupts discussed in this document by wrapping them into dedicated functions:

- **Event detection**: There are several events defined in `uart_event_type_t` that may be reported to a user application using the FreeRTOS queue functionality. You can enable this functionality when calling `uart_driver_install()` described in *Driver Installation*. An example of using Event detection can be found in peripherals/uart/uart_events.
- **FIFO space threshold or transmission timeout reached**: The Tx and Rx FIFO buffers can trigger an interrupt when they are filled with a specific number of characters, or on a timeout of sending or receiving data. To use these interrupts, do the following:
  - Configure respective threshold values of the buffer length and timeout by entering them in the structure `uart_intr_config_t` and calling `uart_intr_config()`
  - Enable the interrupts using the functions `uart_enable_tx_intr()` and `uart_enable_rx_intr()`
  - Disable these interrupts using the corresponding functions `uart_disable_tx_intr()` or `uart_disable_rx_intr()`
- **Pattern detection**: An interrupt triggered on detecting a 'pattern' of the same character being received/sent repeatedly for a number of times. This functionality is demonstrated in the example peripherals/uart/uart_events. It can be used, e.g., to detect a command string followed by a specific number of identical characters (the 'pattern') added at the end of the command string. The following functions are available:
  - Configure and enable this interrupt using `uart_enable_pattern_det_baud_intr()`
  - Disable the interrupt using `uart_disable_pattern_det_intr()`

Macros The API also defines several macros. For example, `UART_FIFO_LEN` defines the length of hardware FIFO buffers; `UART_BITRATE_MAX` gives the maximum baud rate supported by the UART controllers, etc.

Deleting a Driver If the communication established with `uart_driver_install()` is no longer required, the driver can be removed to free allocated resources by calling `uart_driver_delete()`.

Overview of RS485 specific communication options

Note: The following section will use `[UART_REGISTER_NAME].[UART_FIELD_BIT]` to refer to UART register fields/bits. For more information on a specific option bit, see *ESP32 Technical Reference Manual > UART Controller (UART) > Register Summary* [PDF]. Use the register name to navigate to the register description and then find the field/bit.

- `UART_RS485_CONF_REG.UART_RS485_EN`: setting this bit enables RS485 communication mode support.
- `UART_RS485_CONF_REG.UART_RS485TX_RX_EN`: if this bit is set, the transmitter's output signal loops back to the receiver's input signal.
- `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN`: if this bit is set, the transmitter will still be sending data if the receiver is busy (remove collisions automatically by hardware).

The ESP32's RS485 UART hardware can detect signal collisions during transmission of a datagram and generate the interrupt `UART_RS485_CLASH_INT` if this interrupt is enabled. The term collision means that a transmitted datagram is not equal to the one received on the other end. Data collisions are usually associated with the presence of other active devices on the bus or might occur due to bus errors.

The collision detection feature allows handling collisions when their interrupts are activated and triggered. The interrupts `UART_RS485_FRM_ERR_INT` and `UART_RS485_PARITY_ERR_INT` can be used with the collision detection feature to control frame errors and parity bit errors accordingly in RS485 mode. This functionality is

supported in the UART driver and can be used by selecting the *UART_MODE_RS485_APP_CTRL* mode (see the function *uart_set_mode()*).

The collision detection feature can work with circuit A and circuit C (see Section *Interface Connection Options*). In the case of using circuit A or B, the RTS pin connected to the DE pin of the bus driver should be controlled by the user application. Use the function *uart_get_collision_flag()* to check if the collision detection flag has been raised.

The ESP32 UART controllers themselves do not support half-duplex communication as they cannot provide automatic control of the RTS pin connected to the ~RE/DE input of RS485 bus driver. However, half-duplex communication can be achieved via software control of the RTS pin by the UART driver. This can be enabled by selecting the *UART_MODE_RS485_HALF_DUPLEX* mode when calling *uart_set_mode()*.

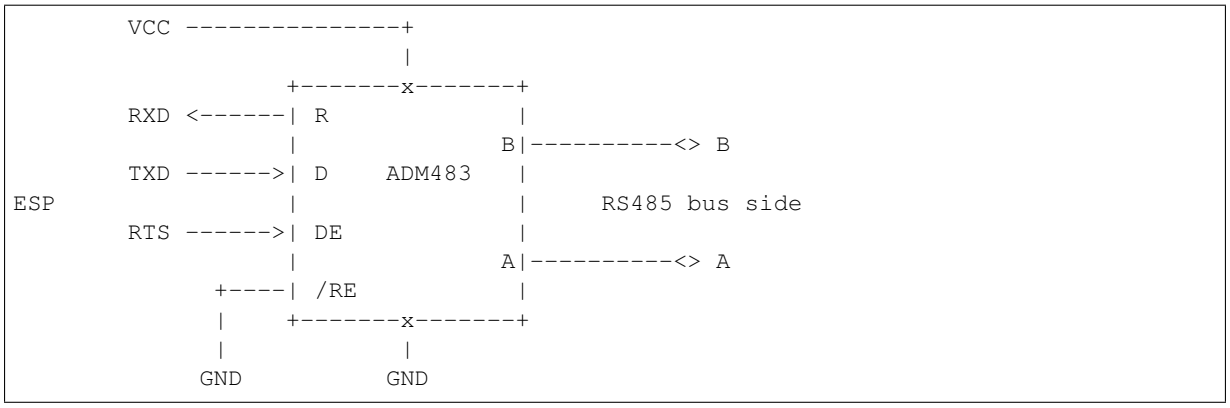
Once the host starts writing data to the Tx FIFO buffer, the UART driver automatically asserts the RTS pin (logic 1); once the last bit of the data has been transmitted, the driver de-asserts the RTS pin (logic 0). To use this mode, the software would have to disable the hardware flow control function. This mode works with all the used circuits shown below.

Interface Connection Options This section provides example schematics to demonstrate the basic aspects of ESP32's RS485 interface connection.

Note:

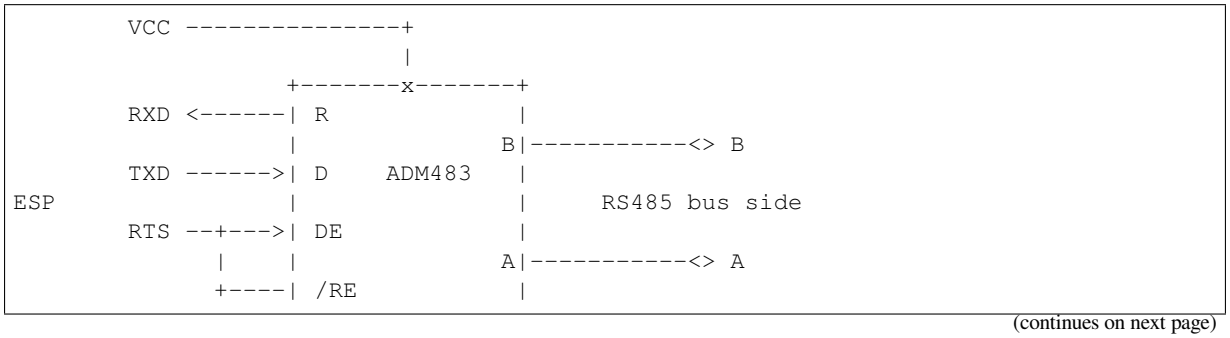
- The schematics below do **not** necessarily contain **all required elements**.
- The **analog devices** ADM483 & ADM2483 are examples of common RS485 transceivers and **can be replaced** with other similar transceivers.

Circuit A: Collision Detection Circuit



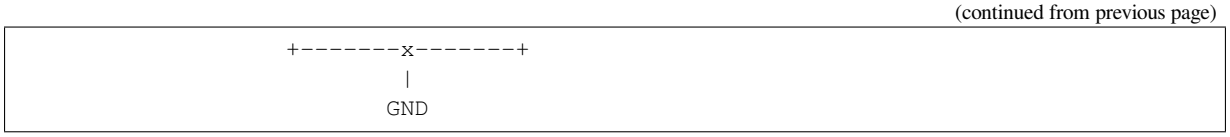
This circuit is preferable because it allows for collision detection and is quite simple at the same time. The receiver in the line driver is constantly enabled, which allows the UART to monitor the RS485 bus. Echo suppression is performed by the UART peripheral when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is enabled.

Circuit B: Manual Switching Transmitter/Receiver Without Collision Detection



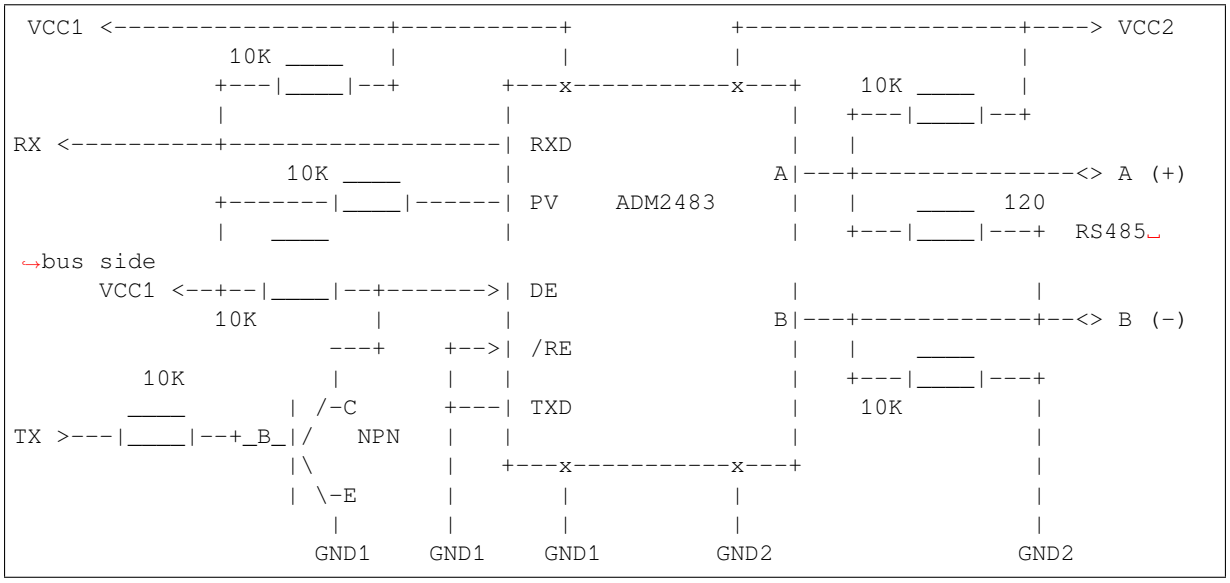
(continues on next page)

(continued from previous page)



This circuit does not allow for collision detection. It suppresses the null bytes that the hardware receives when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is set. The bit `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` is not applicable in this case.

Circuit C: Auto Switching Transmitter/Receiver



This galvanically isolated circuit does not require RTS pin control by a software application or driver because it controls the transceiver direction automatically. However, it requires suppressing null bytes during transmission by setting `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` to 1 and `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` to 0. This setup can work in any RS485 UART mode or even in *`UART_MODE_UART`*.

Application Examples

The table below describes the code examples available in the directory peripherals/uart/.

Code Example	Description
peripherals/uart/uart_echo	Configuring UART settings, installing the UART driver, and reading/writing over the UART1 interface.
peripherals/uart/uart_events	Reporting various communication events, using pattern detection interrupts.
peripherals/uart/uart_async_rxtxtasks	Transmitting and receiving data in two separate FreeRTOS tasks over the same UART.
peripherals/uart/uart_select	Using synchronous I/O multiplexing for UART file descriptors.
peripherals/uart/uart_echo_rs485	Setting up UART driver to communicate over RS485 interface in half-duplex mode. This example is similar to peripherals/uart/uart_echo but allows communication through an RS485 interface chip connected to ESP32 pins.
peripherals/uart/nmea0183_parser	Obtaining GPS information by parsing NMEA0183 statements received from GPS via the UART peripheral.

API Reference

Header File

- components/driver/include/driver/uart.h

Functions

esp_err_t **uart_driver_install** (*uart_port_t* uart_num, int rx_buffer_size, int tx_buffer_size, int queue_size, *QueueHandle_t* *uart_queue, int intr_alloc_flags)

Install UART driver and set the UART to the default configuration.

UART ISR handler will be attached to the same CPU core that this function is running on.

Note: Rx_buffer_size should be greater than UART_FIFO_LEN. Tx_buffer_size should be either zero or greater than UART_FIFO_LEN.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **rx_buffer_size** –UART RX ring buffer size.
- **tx_buffer_size** –UART TX ring buffer size. If set to zero, driver will not use TX buffer, TX function will block task until all data have been sent out.
- **queue_size** –UART event queue size/depth.
- **uart_queue** –UART event queue handle (out param). On success, a new queue handle is written here to provide access to UART events. If set to NULL, driver will not use an event queue.
- **intr_alloc_flags** –Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info. Do not set ESP_INTR_FLAG_IRAM here (the driver's ISR handler is not located in IRAM)

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_driver_delete** (*uart_port_t* uart_num)

Uninstall UART driver.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

bool **uart_is_driver_installed** (*uart_port_t* uart_num)

Checks whether the driver is installed or not.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- true driver is installed
- false driver is not installed

esp_err_t **uart_set_word_length** (*uart_port_t* uart_num, *uart_word_length_t* data_bit)

Set UART data bits.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **data_bit** –UART data bits

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_get_word_length** (*uart_port_t* uart_num, *uart_word_length_t* *data_bit)

Get the UART data bit configuration.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

- **data_bit** –Pointer to accept value of UART data bits.

Returns

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*data_bit)

esp_err_t **uart_set_stop_bits**(*uart_port_t* uart_num, *uart_stop_bits_t* stop_bits)

Set UART stop bits.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **stop_bits** –UART stop bits

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **uart_get_stop_bits**(*uart_port_t* uart_num, *uart_stop_bits_t* *stop_bits)

Get the UART stop bit configuration.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **stop_bits** –Pointer to accept value of UART stop bits.

Returns

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*stop_bit)

esp_err_t **uart_set_parity**(*uart_port_t* uart_num, *uart_parity_t* parity_mode)

Set UART parity mode.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **parity_mode** –the enum of uart parity configuration

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **uart_get_parity**(*uart_port_t* uart_num, *uart_parity_t* *parity_mode)

Get the UART parity mode configuration.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **parity_mode** –Pointer to accept value of UART parity mode.

Returns

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*parity_mode)

esp_err_t **uart_get_sclk_freq**(*uart_sclk_t* sclk, uint32_t *out_freq_hz)

Get the frequency of a clock source for the UART.

Parameters

- **sclk** –Clock source
- **out_freq_hz** –**[out]** Output of frequency, in Hz

Returns

- ESP_ERR_INVALID_ARG: if the clock source is not supported
- otherwise ESP_OK

esp_err_t **uart_set_baudrate**(*uart_port_t* uart_num, uint32_t baudrate)

Set UART baud rate.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **baudrate** –UART baud rate.

Returns

- ESP_FAIL Parameter error

- ESP_OK Success

esp_err_t **uart_get_baudrate**(*uart_port_t* uart_num, uint32_t *baudrate)

Get the UART baud rate configuration.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **baudrate** –Pointer to accept value of UART baud rate

Returns
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*baudrate)

esp_err_t **uart_set_line_inverse**(*uart_port_t* uart_num, uint32_t inverse_mask)

Set UART line inverse mode.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **inverse_mask** –Choose the wires that need to be inverted. Using the ORred mask of `uart_signal_inv_t`

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_hw_flow_ctrl**(*uart_port_t* uart_num, *uart_hw_flowcontrol_t* flow_ctrl, uint8_t rx_thresh)

Set hardware flow control.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **flow_ctrl** –Hardware flow control mode
- **rx_thresh** –Threshold of Hardware RX flow control (0 ~ UART_FIFO_LEN). Only when UART_HW_FLOWCTRL_RTS is set, will the rx_thresh value be set.

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_sw_flow_ctrl**(*uart_port_t* uart_num, bool enable, uint8_t rx_thresh_xon, uint8_t rx_thresh_xoff)

Set software flow control.

Parameters
- **uart_num** –UART_NUM_0, UART_NUM_1 or UART_NUM_2
- **enable** –switch on or off
- **rx_thresh_xon** –low water mark
- **rx_thresh_xoff** –high water mark

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_get_hw_flow_ctrl**(*uart_port_t* uart_num, *uart_hw_flowcontrol_t* *flow_ctrl)

Get the UART hardware flow control configuration.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **flow_ctrl** –Option for different flow control mode.

Returns
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*flow_ctrl)

esp_err_t **uart_clear_intr_status**(*uart_port_t* uart_num, uint32_t clr_mask)

Clear UART interrupt status.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **clr_mask** –Bit mask of the interrupt status to be cleared.

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_enable_intr_mask**(*uart_port_t* uart_num, uint32_t enable_mask)

Set UART interrupt enable.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **enable_mask** –Bit mask of the enable bits.

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_disable_intr_mask**(*uart_port_t* uart_num, uint32_t disable_mask)

Clear UART interrupt enable bits.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **disable_mask** –Bit mask of the disable bits.

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_enable_rx_intr**(*uart_port_t* uart_num)

Enable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_disable_rx_intr**(*uart_port_t* uart_num)

Disable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_disable_tx_intr**(*uart_port_t* uart_num)

Disable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Parameters **uart_num** –UART port number

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_enable_tx_intr**(*uart_port_t* uart_num, int enable, int thresh)

Enable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **enable** –1: enable; 0: disable
- **thresh** –Threshold of TX interrupt, 0 ~ UART_FIFO_LEN

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_pin**(*uart_port_t* uart_num, int tx_io_num, int rx_io_num, int rts_io_num, int cts_io_num)

Assign signals of a UART peripheral to GPIO pins.

Note: If the GPIO number configured for a UART signal matches one of the IOMUX signals for that GPIO, the signal will be connected directly via the IOMUX. Otherwise the GPIO and signal will be connected via the GPIO Matrix. For example, if on an ESP32 the call `uart_set_pin(0, 1, 3, -1, -1)` is performed, as GPIO1 is UART0' s default TX pin and GPIO3 is UART0' s default RX pin, both will be connected to respectively U0TXD and U0RXD through the IOMUX, totally bypassing the GPIO matrix. The check is performed on a per-pin basis. Thus, it is possible to have RX pin binded to a GPIO through the GPIO matrix, whereas TX is binded to its GPIO through the IOMUX.

Note: Internal signal can be output to multiple GPIO pads. Only one GPIO pad can connect with input signal.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **tx_io_num** –UART TX pin GPIO number.
- **rx_io_num** –UART RX pin GPIO number.
- **rts_io_num** –UART RTS pin GPIO number.
- **cts_io_num** –UART CTS pin GPIO number.

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_rts** (*uart_port_t* uart_num, int level)

Manually set the UART RTS pin level.

Note: UART must be configured with hardware flow control disabled.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **level** –1: RTS output low (active); 0: RTS output high (block)

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_dtr** (*uart_port_t* uart_num, int level)

Manually set the UART DTR pin level.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **level** –1: DTR output low; 0: DTR output high

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_tx_idle_num** (*uart_port_t* uart_num, uint16_t idle_num)

Set UART idle interval after tx FIFO is empty.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **idle_num** –idle interval after tx FIFO is empty(unit: the time it takes to send one bit under current baudrate)

Returns

- ESP_OK Success

- ESP_FAIL Parameter error

esp_err_t **uart_param_config** (*uart_port_t* uart_num, const *uart_config_t* *uart_config)

Set UART configuration parameters.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **uart_config** –UART parameter settings

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_intr_config** (*uart_port_t* uart_num, const *uart_intr_config_t* *intr_conf)

Configure UART interrupts.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **intr_conf** –UART interrupt settings

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_wait_tx_done** (*uart_port_t* uart_num, TickType_t ticks_to_wait)

Wait until UART TX FIFO is empty.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **ticks_to_wait** –Timeout, count in RTOS ticks

Returns
- ESP_OK Success
- ESP_FAIL Parameter error
- ESP_ERR_TIMEOUT Timeout

int **uart_tx_chars** (*uart_port_t* uart_num, const char *buffer, uint32_t len)

Send data to the UART port from a given buffer and length.

This function will not wait for enough space in TX FIFO. It will just fill the available TX FIFO and return when the FIFO is full.

Note: This function should only be used when UART TX buffer is not enabled.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **buffer** –data buffer address
- **len** –data length to send

Returns
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

int **uart_write_bytes** (*uart_port_t* uart_num, const void *src, size_t size)

Send data to the UART port from a given buffer and length,.

If the UART driver's parameter 'tx_buffer_size' is set to zero: This function will not return until all the data have been sent out, or at least pushed into TX FIFO.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **src** –data buffer address

- **size** –data length to send

Returns

- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

int **uart_write_bytes_with_break**(*uart_port_t* uart_num, const void *src, size_t size, int brk_len)

Send data to the UART port from a given buffer and length,.

If the UART driver's parameter 'tx_buffer_size' is set to zero: This function will not return until all the data and the break signal have been sent out. After all data is sent out, send a break signal.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually. After all data sent out, send a break signal.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **src** –data buffer address
- **size** –data length to send
- **brk_len** –break signal duration(unit: the time it takes to send one bit at current baudrate)

Returns

- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

int **uart_read_bytes**(*uart_port_t* uart_num, void *buf, uint32_t length, TickType_t ticks_to_wait)

UART read bytes from UART buffer.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **buf** –pointer to the buffer.
- **length** –data length
- **ticks_to_wait** –sTimeout, count in RTOS ticks

Returns

- (-1) Error
- OTHERS (>=0) The number of bytes read from UART FIFO

esp_err_t **uart_flush**(*uart_port_t* uart_num)

Alias of uart_flush_input. UART ring buffer flush. This will discard all data in the UART RX buffer.

Note: Instead of waiting the data sent out, this function will clear UART rx buffer. In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_flush_input**(*uart_port_t* uart_num)

Clear input buffer, discard all the data is in the ring-buffer.

Note: In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_get_buffered_data_len**(*uart_port_t* uart_num, size_t *size)

UART get RX ring buffer cached data length.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **size** –Pointer of size_t to accept cached data length

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_get_tx_buffer_free_size**(*uart_port_t* uart_num, size_t *size)

UART get TX ring buffer free space size.

Parameters
- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **size** –Pointer of size_t to accept the free space size

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **uart_disable_pattern_det_intr**(*uart_port_t* uart_num)

UART disable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detects a series of one same character, the interrupt will be triggered.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_enable_pattern_det_baud_intr**(*uart_port_t* uart_num, char pattern_chr, uint8_t chr_num, int chr_tout, int post_idle, int pre_idle)

UART enable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detect a series of one same character, the interrupt will be triggered.

Parameters
- **uart_num** –UART port number.
- **pattern_chr** –character of the pattern.
- **chr_num** –number of the character, 8bit value.
- **chr_tout** –timeout of the interval between each pattern characters, 16bit value, unit is the baud-rate cycle you configured. When the duration is more than this value, it will not take this data as at_cmd char.
- **post_idle** –idle time after the last pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take the previous data as the last at_cmd char
- **pre_idle** –idle time before the first pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take this data as the first at_cmd char.

Returns
- ESP_OK Success
- ESP_FAIL Parameter error

int **uart_pattern_pop_pos**(*uart_port_t* uart_num)

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, this function will dequeue the first pattern position and move the pointer to next pattern position.

The following APIs will modify the pattern position info: uart_flush_input, uart_read_bytes, uart_driver_delete, uart_pop_pattern_pos It is the application’s responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note: If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

int **uart_pattern_get_pos**(*uart_port_t* uart_num)

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, This function do nothing to the queue.

The following APIs will modify the pattern position info: uart_flush_input, uart_read_bytes, uart_driver_delete, uart_pop_pattern_pos It is the application' s responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note: If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Parameters **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).

Returns

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

esp_err_t **uart_pattern_queue_reset**(*uart_port_t* uart_num, int queue_length)

Allocate a new memory with the given length to save record the detected pattern position in rx buffer.

Parameters

- **uart_num** –UART port number, the max port number is (UART_NUM_MAX -1).
- **queue_length** –Max queue length for the detected pattern. If the queue length is not large enough, some pattern positions might be lost. Set this value to the maximum number of patterns that could be saved in data buffer at the same time.

Returns

- ESP_ERR_NO_MEM No enough memory
- ESP_ERR_INVALID_STATE Driver not installed
- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **uart_set_mode**(*uart_port_t* uart_num, *uart_mode_t* mode)

UART set communication mode.

Note: This function must be executed after uart_driver_install(), when the driver object is initialized.

Parameters

- **uart_num** –Uart number to configure, the max port number is (UART_NUM_MAX -1).
- **mode** –UART UART mode to set

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **uart_set_rx_full_threshold** (*uart_port_t* uart_num, int threshold)

Set uart threshold value for RX fifo full.

Note: If application is using higher baudrate and it is observed that bytes in hardware RX fifo are overwritten then this threshold can be reduced

---

Parameters

- **uart_num** –UART_NUM_0, UART_NUM_1 or UART_NUM_2
- **threshold** –Threshold value above which RX fifo full interrupt is generated

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

esp_err_t **uart_set_tx_empty_threshold** (*uart_port_t* uart_num, int threshold)

Set uart threshold values for TX fifo empty.

Parameters

- **uart_num** –UART_NUM_0, UART_NUM_1 or UART_NUM_2
- **threshold** –Threshold value below which TX fifo empty interrupt is generated

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

esp_err_t **uart_set_rx_timeout** (*uart_port_t* uart_num, const uint8_t tout_thresh)

UART set threshold timeout for TOUT feature.

Parameters

- **uart_num** –Uart number to configure, the max port number is (UART_NUM_MAX -1).
- **tout_thresh** –This parameter defines timeout threshold in uart symbol periods. The maximum value of threshold is 126. tout_thresh = 1, defines TOUT interrupt timeout equal to transmission time of one symbol (~11 bit) on current baudrate. If the time is expired the UART_RXFIFO_TOUT_INT interrupt is triggered. If tout_thresh == 0, the TOUT feature is disabled.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

esp_err_t **uart_get_collision_flag** (*uart_port_t* uart_num, bool *collision_flag)

Returns collision detection flag for RS485 mode Function returns the collision detection flag into variable pointed by collision_flag. *collision_flag = true, if collision detected else it is equal to false. This function should be executed when actual transmission is completed (after uart_write_bytes()).

Parameters

- **uart_num** –Uart number to configure the max port number is (UART_NUM_MAX -1).
- **collision_flag** –Pointer to variable of type bool to return collision flag.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **uart_set_wakeup_threshold** (*uart_port_t* uart_num, int wakeup_threshold)

Set the number of RX pin signal edges for light sleep wakeup.

UART can be used to wake up the system from light sleep. This feature works by counting the number of positive edges on RX pin and comparing the count to the threshold. When the count exceeds the threshold, system is woken up from light sleep. This function allows setting the threshold value.

Stop bit and parity bits (if enabled) also contribute to the number of edges. For example, letter ‘a’ with ASCII code 97 is encoded as 0100001101 on the wire (with 8n1 configuration), start and stop bits included. This sequence has 3 positive edges (transitions from 0 to 1). Therefore, to wake up the system when ‘a’ is sent, set wakeup_threshold=3.

The character that triggers wakeup is not received by UART (i.e. it can not be obtained from UART FIFO). Depending on the baud rate, a few characters after that will also not be received. Note that when the chip enters and exits light sleep mode, APB frequency will be changing. To make sure that UART has correct baud rate all the time, select UART_SCLK_REF_TICK or UART_SCLK_XTAL as UART clock source in *uart_config_t::source_clk*.

Note: in ESP32, the wakeup signal can only be input via IO_MUX (i.e. GPIO3 should be configured as function_1 to wake up UART0, GPIO9 should be configured as function_5 to wake up UART1), UART2 does not support light sleep wakeup feature.

Parameters
- **`uart_num`** –UART number, the max port number is (UART_NUM_MAX -1).
- **`wakeup_threshold`** –number of RX edges for light sleep wakeup, value is 3 .. 0x3ff.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if uart_num is incorrect or wakeup_threshold is outside of [3, 0x3ff] range.

esp_err_t **`uart_get_wakeup_threshold`**(*uart_port_t* uart_num, int *out_wakeup_threshold)

Get the number of RX pin signal edges for light sleep wakeup.

See description of uart_set_wakeup_threshold for the explanation of UART wakeup feature.

Parameters
- **`uart_num`** –UART number, the max port number is (UART_NUM_MAX -1).
- **`out_wakeup_threshold`** –**[out]** output, set to the current value of wakeup threshold for the given UART.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_wakeup_threshold is NULL

esp_err_t **`uart_wait_tx_idle_polling`**(*uart_port_t* uart_num)

Wait until UART tx memory empty and the last char send ok (polling mode).

-

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters **`uart_num`** –UART number

esp_err_t **`uart_set_loop_back`**(*uart_port_t* uart_num, bool loop_back_en)

Configure TX signal loop back to RX module, just for the test usage.

-

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters

- **uart_num** –UART number
- **loop_back_en** –Set ture to enable the loop back function, else set it false.

void **uart_set_always_rx_timeout**(*uart_port_t* uart_num, bool always_rx_timeout_en)

Configure behavior of UART RX timeout interrupt.

When always_rx_timeout is true, timeout interrupt is triggered even if FIFO is full. This function can cause extra timeout interrupts triggered only to send the timeout event. Call this function only if you want to ensure timeout interrupt will always happen after a byte stream.

Parameters

- **uart_num** –UART number
- **always_rx_timeout_en** –Set to false enable the default behavior of timeout interrupt, set it to true to always trigger timeout interrupt.

Structures

struct **uart_intr_config_t**

UART interrupt configuration parameters for uart_intr_config function.

Public Members

uint32_t **intr_enable_mask**

UART interrupt enable mask, choose from UART_XXXX_INT_ENA_M under UART_INT_ENA_REG(i), connect with bit-or operator

uint8_t **rx_timeout_thresh**

UART timeout interrupt threshold (unit: time of sending one byte)

uint8_t **txfifo_empty_intr_thresh**

UART TX empty interrupt threshold.

uint8_t **rxfifo_full_thresh**

UART RX full interrupt threshold.

struct **uart_event_t**

Event structure used in UART event queue.

Public Members

uart_event_type_t **type**

UART event type

size_t **size**

UART data size for UART_DATA event

bool **timeout_flag**

UART data read timeout flag for UART_DATA event (no new data received during configured RX TOUT) If the event is caused by FIFO-full interrupt, then there will be no event with the timeout flag before the next byte coming.

Macros

UART_NUM_0

UART port 0

UART_NUM_1

UART port 1

UART_NUM_2

UART port 2

UART_NUM_MAX

UART port max

UART_PIN_NO_CHANGE

UART_FIFO_LEN

Length of the UART HW FIFO.

UART_BITRATE_MAX

Maximum configurable bitrate.

Type Definitions

typedef *intr_handle_t* **uart_isr_handle_t**

Enumerations

enum **uart_event_type_t**

UART event types used in the ring buffer.

Values:

enumerator **UART_DATA**

UART data event

enumerator **UART_BREAK**

UART break event

enumerator **UART_BUFFER_FULL**

UART RX buffer full event

enumerator **UART_FIFO_OVF**

UART FIFO overflow event

enumerator **UART_FRAME_ERR**

UART RX frame error event

enumerator **UART_PARITY_ERR**

UART RX parity event

enumerator **UART_DATA_BREAK**

UART TX data and break event

enumerator **UART_PATTERN_DET**

UART pattern detected

enumerator **UART_EVENT_MAX**

UART event max index

Header File

- components/hal/include/hal/uart_types.h

Structures

struct **uart_at_cmd_t**

UART AT cmd char configuration parameters Note that this function may different on different chip. Please refer to the TRM at confirguration.

Public Members

uint8_t **cmd_char**

UART AT cmd char

uint8_t **char_num**

AT cmd char repeat number

uint32_t **gap_tout**

gap time(in baud-rate) between AT cmd char

uint32_t **pre_idle**

the idle time(in baud-rate) between the non AT char and first AT char

uint32_t **post_idle**

the idle time(in baud-rate) between the last AT char and the none AT char

struct **uart_sw_flowctrl_t**

UART software flow control configuration parameters.

Public Members

uint8_t **xon_char**

Xon flow control char

uint8_t **xoff_char**

Xoff flow control char

uint8_t **xon_thrd**

If the software flow control is enabled and the data amount in rxfifo is less than xon_thrd, an xon_char will be sent

uint8_t **xoff_thrd**

If the software flow control is enabled and the data amount in rxfifo is more than xoff_thrd, an xoff_char will be sent

struct **uart_config_t**

UART configuration parameters for uart_param_config function.

Public Members

int **baud_rate**

UART baud rate

uart_word_length_t **data_bits**

UART byte size

uart_parity_t **parity**

UART parity mode

uart_stop_bits_t **stop_bits**

UART stop bits

uart_hw_flowcontrol_t **flow_ctrl**

UART HW flow control mode (cts/rts)

uint8_t **rx_flow_ctrl_thresh**

UART HW RTS threshold

uart_sclk_t **source_clk**

UART source clock selection

Type Definitions

typedef int **uart_port_t**

UART port number, can be UART_NUM_0 ~ (UART_NUM_MAX -1).

typedef *soc_periph_uart_clk_src_legacy_t* **uart_sclk_t**

UART source clock.

Enumerations

enum **uart_mode_t**

UART mode selection.

Values:

enumerator **UART_MODE_UART**

mode: regular UART mode

enumerator **UART_MODE_RS485_HALF_DUPLEX**

mode: half duplex RS485 UART mode control by RTS pin

enumerator **UART_MODE_IRDA**

mode: IRDA UART mode

enumerator **UART_MODE_RS485_COLLISION_DETECT**

mode: RS485 collision detection UART mode (used for test purposes)

enumerator **UART_MODE_RS485_APP_CTRL**

mode: application control RS485 UART mode (used for test purposes)

enum **uart_word_length_t**

UART word length constants.

Values:

enumerator **UART_DATA_5_BITS**

word length: 5bits

enumerator **UART_DATA_6_BITS**

word length: 6bits

enumerator **UART_DATA_7_BITS**

word length: 7bits

enumerator **UART_DATA_8_BITS**

word length: 8bits

enumerator **UART_DATA_BITS_MAX**

enum **uart_stop_bits_t**

UART stop bits number.

Values:

enumerator **UART_STOP_BITS_1**

stop bit: 1bit

enumerator **UART_STOP_BITS_1_5**

stop bit: 1.5bits

enumerator **UART_STOP_BITS_2**

stop bit: 2bits

enumerator **UART_STOP_BITS_MAX**

enum **uart_parity_t**

UART parity constants.

Values:

enumerator **UART_PARITY_DISABLE**

Disable UART parity

enumerator **UART_PARITY_EVEN**

Enable UART even parity

enumerator **UART_PARITY_ODD**

Enable UART odd parity

enum **uart_hw_flowcontrol_t**

UART hardware flow control modes.

Values:

enumerator **UART_HW_FLOWCTRL_DISABLE**

disable hardware flow control

enumerator **UART_HW_FLOWCTRL_RTS**

enable RX hardware flow control (rts)

enumerator **UART_HW_FLOWCTRL_CTS**

enable TX hardware flow control (cts)

enumerator **UART_HW_FLOWCTRL_CTS_RTS**

enable hardware flow control

enumerator **UART_HW_FLOWCTRL_MAX**

enum **uart_signal_inv_t**

UART signal bit map.

Values:

enumerator **UART_SIGNAL_INV_DISABLE**

Disable UART signal inverse

enumerator **UART_SIGNAL_IRDA_TX_INV**

inverse the UART irda_tx signal

enumerator **UART_SIGNAL_IRDA_RX_INV**

inverse the UART irda_rx signal

enumerator **UART_SIGNAL_RXD_INV**

inverse the UART rxd signal

enumerator **UART_SIGNAL_CTS_INV**

inverse the UART cts signal

enumerator **UART_SIGNAL_DSR_INV**

inverse the UART dsr signal

enumerator **UART_SIGNAL_TXD_INV**

inverse the UART txd signal

enumerator **UART_SIGNAL_RTS_INV**

inverse the UART rts signal

enumerator **UART_SIGNAL_DTR_INV**

inverse the UART dtr signal

GPIO Lookup Macros The UART peripherals have dedicated IO_MUX pins to which they are connected directly. However, signals can also be routed to other pins using the less direct GPIO matrix. To use direct routes, you need to know which pin is a dedicated IO_MUX pin for a UART channel. GPIO Lookup Macros simplify the process of finding and assigning IO_MUX pins. You choose a macro based on either the IO_MUX pin number, or a required UART channel name, and the macro will return the matching counterpart for you. See some examples below.

Note: These macros are useful if you need very high UART baud rates (over 40 MHz), which means you will have to use IO_MUX pins only. In other cases, these macros can be ignored, and you can use the GPIO Matrix as it allows you to configure any GPIO pin for any UART function.

---

1. *UART_NUM_2_TXD_DIRECT_GPIO_NUM* returns the IO_MUX pin number of UART channel 2 TXD pin (pin 17)
2. *UART_GPIO19_DIRECT_CHANNEL* returns the UART number of GPIO 19 when connected to the UART peripheral via IO_MUX (this is UART_NUM_0)
3. *UART_CTS_GPIO19_DIRECT_CHANNEL* returns the UART number of GPIO 19 when used as the UART CTS pin via IO_MUX (this is UART_NUM_0). Similar to the above macro but specifies the pin function which is also part of the IO_MUX assignment.

Header File

- components/soc/esp32/include/soc/uart_channel.h

Macros

UART_GPIO1_DIRECT_CHANNEL

UART_NUM_0_TXD_DIRECT_GPIO_NUM

UART_GPIO3_DIRECT_CHANNEL

UART_NUM_0_RXD_DIRECT_GPIO_NUM

UART_GPIO19_DIRECT_CHANNEL

UART_NUM_0_CTS_DIRECT_GPIO_NUM

UART_GPIO22_DIRECT_CHANNEL

UART_NUM_0_RTS_DIRECT_GPIO_NUM

UART_TXD_GPIO1_DIRECT_CHANNEL

UART_RXD_GPIO3_DIRECT_CHANNEL

UART_CTS_GPIO19_DIRECT_CHANNEL

UART_RTS_GPIO22_DIRECT_CHANNEL

UART_GPIO10_DIRECT_CHANNEL

UART_NUM_1_TXD_DIRECT_GPIO_NUM

UART_GPIO9_DIRECT_CHANNEL

UART_NUM_1_RXD_DIRECT_GPIO_NUM

UART_GPIO6_DIRECT_CHANNEL

UART_NUM_1_CTS_DIRECT_GPIO_NUM

UART_GPIO11_DIRECT_CHANNEL

UART_NUM_1_RTS_DIRECT_GPIO_NUM

UART_TXD_GPIO10_DIRECT_CHANNEL

UART_RXD_GPIO9_DIRECT_CHANNEL

UART_CTS_GPIO6_DIRECT_CHANNEL

UART_RTS_GPIO11_DIRECT_CHANNEL

UART_GPIO17_DIRECT_CHANNEL

UART_NUM_2_TXD_DIRECT_GPIO_NUM

UART_GPIO16_DIRECT_CHANNEL

UART_NUM_2_RXD_DIRECT_GPIO_NUM

UART_GPIO8_DIRECT_CHANNEL

UART_NUM_2_CTS_DIRECT_GPIO_NUM

UART_GPIO7_DIRECT_CHANNEL

UART_NUM_2_RTS_DIRECT_GPIO_NUM

UART_TXD_GPIO17_DIRECT_CHANNEL

UART_RXD_GPIO16_DIRECT_CHANNEL

UART_CTS_GPIO8_DIRECT_CHANNEL

UART_RTS_GPIO7_DIRECT_CHANNEL

Code examples for this API section are provided in the peripherals directory of ESP-IDF examples.

2.7 Project Configuration

2.7.1 Introduction

ESP-IDF uses kconfiglib which is a Python-based extension to the Kconfig system which provides a compile-time project configuration mechanism. Kconfig is based around options of several types: integer, string, boolean. Kconfig files specify dependencies between options, default values of the options, the way the options are grouped together, etc.

For the complete list of available features please see Kconfig and kconfiglib extentions.

2.7.2 Project Configuration Menu

Application developers can open a terminal-based project configuration menu with the `idf.py menuconfig` build target.

After being updated, this configuration is saved inside `sdkconfig` file in the project root directory. Based on `sdkconfig`, application build targets will generate `sdkconfig.h` file in the build directory, and will make sdkconfig options available to the project build system and source files.

2.7.3 Using sdkconfig.defaults

In some cases, such as when `sdkconfig` file is under revision control, the fact that `sdkconfig` file gets changed by the build system may be inconvenient. The build system offers a way to avoid this, in the form of `sdkconfig.defaults` file. This file is never touched by the build system, and can be created manually or automatically. It can contain all the options which matter for the given application and are different from the default ones. The format is the same as that of the `sdkconfig` file. `sdkconfig.defaults` can be created manually when one remembers all the changed configurations. Otherwise, the file can be generated automatically by running the `idf.py save-defconfig` command.

Once `sdkconfig.defaults` is created, `sdkconfig` can be deleted and added to the ignore list of the revision control system (e.g. `.gitignore` file for `git`). Project build targets will automatically create `sdkconfig` file, populated with the settings from `sdkconfig.defaults` file, and the rest of the settings will be set to their default values. Note that the build process will not override settings that are already in `sdkconfig` by ones from `sdkconfig.defaults`. For more information, see *Custom sdkconfig defaults*.

2.7.4 Kconfig Formatting Rules

The following attributes of `Kconfig` files are standardized:

- Within any menu, option names should have a consistent prefix. The prefix length is currently set to at least 3 characters.
- The indentation style is 4 characters created by spaces. All sub-items belonging to a parent item are indented by one level deeper. For example, `menu` is indented by 0 characters, the `config` inside of the menu by 4 characters, the help of the `config` by 8 characters and the text of the `help` by 12 characters.
- No trailing spaces are allowed at the end of the lines.
- The maximum length of options is set to 40 characters.
- The maximum length of lines is set to 120 characters.

Format checker

`tools/ci/check_kconfigs.py` is provided for checking the `Kconfig` formatting rules. The checker checks all `Kconfig` and `Kconfig.projbuild` files in the ESP-IDF directory and generates a new file with suffix `.new` with some recommendations how to fix issues (if there are any). Please note that the checker cannot correct all rules and the responsibility of the developer is to check and make final corrections in order to pass the tests. For example, indentations will be corrected if there isn't some misleading previous formatting but it cannot come up with a common prefix for options inside a menu.

2.7.5 Backward Compatibility of Kconfig Options

The standard Kconfig tools ignore unknown options in `sdkconfig`. So if a developer has custom settings for options which are renamed in newer ESP-IDF releases then the given setting for the option would be silently ignored. Therefore, several features have been adopted to avoid this:

1. `confgen.py` is used by the tool chain to pre-process `sdkconfig` files before anything else, for example `menuconfig`, would read them. As the consequence, the settings for old options will be kept and not ignored.
2. `confgen.py` recursively finds all `sdkconfig.rename` files in ESP-IDF directory which contain old and new `Kconfig` option names. Old options are replaced by new ones in the `sdkconfig` file. Renames that should only appear for a single target can be placed in a target specific rename file: *sdkconfig.rename.TARGET*, where *TARGET* is the target name, e.g. *sdkconfig.rename.esp32s2*.
3. `confgen.py` post-processes `sdkconfig` files and generates all build outputs (`sdkconfig.h`, `sdkconfig.cmake`, `auto.conf`) by adding a list of compatibility statements, i.e. value of the old option is set the value of the new option (after modification). This is done in order to not break customer codes where old option might still be used.
4. *Deprecated options and their replacements* are automatically generated by `confgen.py`.

2.7.6 Configuration Options Reference

Subsequent sections contain the list of available ESP-IDF options, automatically generated from Kconfig files. Note that depending on the options selected, some options listed here may not be visible by default in the interface of menuconfig.

By convention, all option names are upper case with underscores. When Kconfig generates sdkconfig and sdkconfig.h files, option names are prefixed with `CONFIG_`. So if an option `ENABLE_FOO` is defined in a Kconfig file and selected in menuconfig, then sdkconfig and sdkconfig.h files will have `CONFIG_ENABLE_FOO` defined. In this reference, option names are also prefixed with `CONFIG_`, same as in the source code.

Build type

Contains:

- *CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS*
- *CONFIG_APP_COMPATIBLE_PRE_V2_1_BOOTLOADERS*

- *CONFIG_APP_BUILD_TYPE*
- *CONFIG_APP_REPRODUCIBLE_BUILD*
- *CONFIG_APP_NO_BLOBS*

CONFIG_APP_BUILD_TYPE

Application build type

Found in: Build type

Select the way the application is built.

By default, the application is built as a binary file in a format compatible with the ESP-IDF bootloader. In addition to this application, 2nd stage bootloader is also built. Application and bootloader binaries can be written into flash and loaded/executed from there.

Another option, useful for only very small and limited applications, is to only link the .elf file of the application, such that it can be loaded directly into RAM over JTAG. Note that since IRAM and DRAM sizes are very limited, it is not possible to build any complex application this way. However for kinds of testing and debugging, this option may provide faster iterations, since the application does not need to be written into flash. Note that at the moment, ESP-IDF does not contain all the startup code required to initialize the CPUs and ROM memory (data/bss). Therefore it is necessary to execute a bit of ROM code prior to executing the application. A gdbinit file may look as follows (for ESP32):

> # Connect to a running instance of OpenOCD target remote :3333 # Reset and halt the target mon reset halt # Run to a specific point in ROM code, # where most of initialization is complete. thb *0x40007d54 c # Load the application into RAM load # Run till app_main tb app_main c

Execute this gdbinit file as follows:

> xtensa-esp32-elf-gdb build/app-name.elf -x gdbinit

Example gdbinit files for other targets can be found in tools/test_apps/system/gdb_loadable_elf/

Recommended sdkconfig.defaults for building loadable ELF files is as follows. CONFIG_APP_BUILD_TYPE_ELF_RAM is required, other options help reduce application memory footprint.

> CONFIG_APP_BUILD_TYPE_ELF_RAM=y CONFIG_VFS_SUPPORT_TERMIOS= CONFIG_NEWLIB_NANO_FORMAT=y CONFIG_ESP_SYSTEM_PANIC_PRINT_HALT=y CONFIG_ESP_DEBUG_STUBS_ENABLE= CONFIG_ESP_ERR_TO_NAME_LOOKUP=

Available options:

- Default (binary application + 2nd stage bootloader) (APP_BUILD_TYPE_APP_2NDBOOT)
- ELF file, loadable into RAM (EXPERIMENTAL)) (APP_BUILD_TYPE_ELF_RAM)

CONFIG_APP_REPRODUCIBLE_BUILD

Enable reproducible build

Found in: Build type

If enabled, all date, time, and path information would be eliminated. A .gdbinit file would be create automatically. (or will be append if you have one already)

Default value:

- No (disabled)

CONFIG_APP_NO_BLOBS

No Binary Blobs

Found in: Build type

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

Default value:

- No (disabled)

CONFIG_APP_COMPATIBLE_PRE_V2_1_BOOTLOADERS

App compatible with bootloaders before ESP-IDF v2.1

Found in: Build type

Bootloaders before ESP-IDF v2.1 did less initialisation of the system clock. This setting needs to be enabled to build an app which can be booted by these older bootloaders.

If this setting is enabled, the app can be booted by any bootloader from IDF v1.0 up to the current version.

If this setting is disabled, the app can only be booted by bootloaders from IDF v2.1 or newer.

Enabling this setting adds approximately 1KB to the app' s IRAM usage.

Default value:

- No (disabled)

CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS

App compatible with bootloader and partition table before ESP-IDF v3.1

Found in: Build type

Partition tables before ESP-IDF V3.1 do not contain an MD5 checksum field, and the bootloader before ESP-IDF v3.1 cannot read a partition table that contains an MD5 checksum field.

Enable this option only if your app needs to boot on a bootloader and/or partition table that was generated from a version *before* ESP-IDF v3.1.

If this option and Flash Encryption are enabled at the same time, and any data partitions in the partition table are marked Encrypted, then the partition encrypted flag should be manually verified in the app before accessing the partition (see CVE-2021-27926).

Default value:

- No (disabled)

Bootloader config

Contains:

- *CONFIG_BOOTLOADER_LOG_LEVEL*
- *CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION*
- *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE*
- *CONFIG_BOOTLOADER_REGION_PROTECTION_ENABLE*
- *CONFIG_BOOTLOADER_APP_TEST*
- *CONFIG_BOOTLOADER_FACTORY_RESET*
- *CONFIG_BOOTLOADER_HOLD_TIME_GPIO*
- *CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC*
- *Serial Flash Configurations*
- *CONFIG_BOOTLOADER_SKIP_VALIDATE_ALWAYS*
- *CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON*
- *CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP*

- *CONFIG_BOOTLOADER_WDT_ENABLE*
- *CONFIG_BOOTLOADER_VDDSDIO_BOOST*

CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION

Bootloader optimization Level

Found in: Bootloader config

This option sets compiler optimization level (gcc -O argument) for the bootloader.

- The default "Size" setting will add the -0s flag to CFLAGS.
- The "Debug" setting will add the -Og flag to CFLAGS.
- The "Performance" setting will add the -O2 flag to CFLAGS.
- The "None" setting will add the -O0 flag to CFLAGS.

Note that custom optimization levels may be unsupported.

Available options:

- Size (-Os) (BOOTLOADER_COMPILER_OPTIMIZATION_SIZE)
- Debug (-Og) (BOOTLOADER_COMPILER_OPTIMIZATION_DEBUG)
- Optimize for performance (-O2) (BOOTLOADER_COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (BOOTLOADER_COMPILER_OPTIMIZATION_NONE)

CONFIG_BOOTLOADER_LOG_LEVEL

Bootloader log verbosity

Found in: Bootloader config

Specify how much output to see in bootloader logs.

Available options:

- No output (BOOTLOADER_LOG_LEVEL_NONE)
- Error (BOOTLOADER_LOG_LEVEL_ERROR)
- Warning (BOOTLOADER_LOG_LEVEL_WARN)
- Info (BOOTLOADER_LOG_LEVEL_INFO)
- Debug (BOOTLOADER_LOG_LEVEL_DEBUG)
- Verbose (BOOTLOADER_LOG_LEVEL_VERBOSE)

Serial Flash Configurations

Contains:

- *CONFIG_BOOTLOADER_FLASH_DC_AWARE*
- *CONFIG_BOOTLOADER_SPI_WP_PIN*
- *CONFIG_BOOTLOADER_FLASH_XMC_SUPPORT*
- *CONFIG_BOOTLOADER_SPI_CUSTOM_WP_PIN*

CONFIG_BOOTLOADER_SPI_CUSTOM_WP_PIN

Use custom SPI Flash WP Pin when flash pins set in eFuse (read help)

Found in: Bootloader config > Serial Flash Configurations

This setting is only used if the SPI flash pins have been overridden by setting the eFuses SPI_PAD_CONFIG_xxx, and the SPI flash mode is QIO or QOUT.

When this is the case, the eFuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin "SD_DATA_3" or SPI flash pin "IO2") is not specified in eFuse. The same pin is also used for external SPIRAM if it is enabled.

If this config item is set to N (default), the correct WP pin will be automatically used for any Espressif chip or module with integrated flash. If a custom setting is needed, set this config item to Y and specify the GPIO number connected to the WP.

Default value:

- No (disabled) if ESPTOOLPY_FLASHMODE_QIO || ESPTOOLPY_FLASHMODE_QOUT

CONFIG_BOOTLOADER_SPI_WP_PIN

Custom SPI Flash WP Pin

Found in: Bootloader config > Serial Flash Configurations

The option "Use custom SPI Flash WP Pin" must be set or this value is ignored

If burning a customized set of SPI flash pins in eFuse and using QIO or QOUT mode for flash, set this value to the GPIO number of the SPI flash WP pin.

Range:

- from 0 to 33 if ESPTOOLPY_FLASHMODE_QIO || ESPTOOLPY_FLASHMODE_QOUT

Default value:

- 7 if ESPTOOLPY_FLASHMODE_QIO || ESPTOOLPY_FLASHMODE_QOUT

CONFIG_BOOTLOADER_FLASH_DC_AWARE

Allow app adjust Dummy Cycle bits in SPI Flash for higher frequency (READ HELP FIRST)

Found in: Bootloader config > Serial Flash Configurations

This will force 2nd bootloader to be loaded by DOUT mode, and will restore Dummy Cycle setting by resetting the Flash

CONFIG_BOOTLOADER_FLASH_XMC_SUPPORT

Enable the support for flash chips of XMC (READ DOCS FIRST)

Found in: Bootloader config > Serial Flash Configurations

Perform the startup flow recommended by XMC. Please consult XMC for the details of this flow. XMC chips will be forbidden to be used, when this option is disabled.

DON'T DISABLE THIS UNLESS YOU KNOW WHAT YOU ARE DOING.

comment "Features below require specific hardware (READ DOCS FIRST!)"

Default value:

- Yes (enabled)

CONFIG_BOOTLOADER_VDDSDIO_BOOST

VDDSDIO LDO voltage

Found in: Bootloader config

If this option is enabled, and VDDSDIO LDO is set to 1.8V (using eFuse or MTDI bootstrapping pin), bootloader will change LDO settings to output 1.9V instead. This helps prevent flash chip from browning out during flash programming operations.

This option has no effect if VDDSDIO is set to 3.3V, or if the internal VDDSDIO regulator is disabled via eFuse.

Available options:

- 1.8V (BOOTLOADER_VDDSDIO_BOOST_1_8V)
- 1.9V (BOOTLOADER_VDDSDIO_BOOST_1_9V)

CONFIG_BOOTLOADER_FACTORY_RESET

GPIO triggers factory reset

Found in: Bootloader config

Allows to reset the device to factory settings: - clear one or more data partitions; - boot from "factory" partition. The factory reset will occur if there is a GPIO input held at the configured level while device starts up. See settings below.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET

Number of the GPIO input for factory reset

Found in: Bootloader config > CONFIG_BOOTLOADER_FACTORY_RESET

The selected GPIO will be configured as an input with internal pull-up enabled (note that on some SoCs. not all pins have an internal pull-up, consult the hardware datasheet for details.) To trigger a factory reset, this GPIO must be held high or low (as configured) on startup.

Range:

- from 0 to 39 if *CONFIG_BOOTLOADER_FACTORY_RESET*

Default value:

- 4 if *CONFIG_BOOTLOADER_FACTORY_RESET*

CONFIG_BOOTLOADER_FACTORY_RESET_PIN_LEVEL

Factory reset GPIO level

Found in: Bootloader config > CONFIG_BOOTLOADER_FACTORY_RESET

Pin level for factory reset, can be triggered on low or high.

Available options:

- Reset on GPIO low (BOOTLOADER_FACTORY_RESET_PIN_LOW)
- Reset on GPIO high (BOOTLOADER_FACTORY_RESET_PIN_HIGH)

CONFIG_BOOTLOADER_OTA_DATA_ERASE

Clear OTA data on factory reset (select factory partition)

Found in: Bootloader config > CONFIG_BOOTLOADER_FACTORY_RESET

The device will boot from "factory" partition (or OTA slot 0 if no factory partition is present) after a factory reset.

CONFIG_BOOTLOADER_DATA_FACTORY_RESET

Comma-separated names of partitions to clear on factory reset

Found in: Bootloader config > CONFIG_BOOTLOADER_FACTORY_RESET

Allows customers to select which data partitions will be erased while factory reset.

Specify the names of partitions as a comma-delimited with optional spaces for readability. (Like this: "nvs, phy_init, …") Make sure that the name specified in the partition table and here are the same. Partitions of type "app" cannot be specified here.

Default value:

- "nvs" if *CONFIG_BOOTLOADER_FACTORY_RESET*

CONFIG_BOOTLOADER_APP_TEST

GPIO triggers boot from test app partition

Found in: Bootloader config

Allows to run the test app from "TEST" partition. A boot from "test" partition will occur if there is a GPIO input pulled low while device starts up. See settings below.

Default value:

- No (disabled) if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*

CONFIG_BOOTLOADER_NUM_PIN_APP_TEST

Number of the GPIO input to boot TEST partition

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_TEST

The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset. After the GPIO input is deactivated and the device reboots, the old application will boot. (factory or OTA[x]). Note that GPIO34-39 do not have an internal pullup and an external one must be provided.

Range:

- from 0 to 39 if *CONFIG_BOOTLOADER_APP_TEST*

Default value:

- 18 if *CONFIG_BOOTLOADER_APP_TEST*

CONFIG_BOOTLOADER_APP_TEST_PIN_LEVEL

App test GPIO level

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_TEST

Pin level for app test, can be triggered on low or high.

Available options:

- Enter test app on GPIO low (BOOTLOADER_APP_TEST_PIN_LOW)
- Enter test app on GPIO high (BOOTLOADER_APP_TEST_PIN_HIGH)

CONFIG_BOOTLOADER_HOLD_TIME_GPIO

Hold time of GPIO for reset/test mode (seconds)

Found in: Bootloader config

The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

Default value:

- 5 if *CONFIG_BOOTLOADER_FACTORY_RESET* || *CONFIG_BOOTLOADER_APP_TEST*

CONFIG_BOOTLOADER_REGION_PROTECTION_ENABLE

Enable protection for unmapped memory regions

Found in: Bootloader config

Protects the unmapped memory regions of the entire address space from unintended accesses. This will ensure that an exception will be triggered whenever the CPU performs a memory operation on unmapped regions of the address space.

Default value:

- Yes (enabled)

CONFIG_BOOTLOADER_WDT_ENABLE

Use RTC watchdog in start code

Found in: Bootloader config

Tracks the execution time of startup code. If the execution time is exceeded, the RTC_WDT will restart system. It is also useful to prevent a lock up in start code caused by an unstable power source. NOTE: Tracks the execution time starts from the bootloader code - re-set timeout, while selecting the source for slow_clk - and ends calling app_main. Re-set timeout is needed due to WDT uses a SLOW_CLK clock source. After changing a frequency slow_clk a time of WDT needs to re-set for new frequency. slow_clk depends on RTC_CLK_SRC (INTERNAL_RC or EXTERNAL_CRYSTAL).

Default value:

- Yes (enabled)

CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE

Allows RTC watchdog disable in user code

Found in: Bootloader config > CONFIG_BOOTLOADER_WDT_ENABLE

If this option is set, the ESP-IDF app must explicitly reset, feed, or disable the rtc_wdt in the app's own code. If this option is not set (default), then rtc_wdt will be disabled by ESP-IDF before calling the app_main() function.

Use function rtc_wdt_feed() for resetting counter of rtc_wdt. Use function rtc_wdt_disable() for disabling rtc_wdt.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_WDT_TIME_MS

Timeout for RTC watchdog (ms)

Found in: Bootloader config > CONFIG_BOOTLOADER_WDT_ENABLE

Verify that this parameter is correct and more then the execution time. Pay attention to options such as reset to factory, trigger test partition and encryption on boot - these options can increase the execution time. Note: RTC_WDT will reset while encryption operations will be performed.

Range:

- from 0 to 120000

Default value:

- 9000

CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE

Enable app rollback support

Found in: Bootloader config

After updating the app, the bootloader runs a new app with the "ESP_OTA_IMG_PENDING_VERIFY" state set. This state prevents the re-run of this app. After the first boot of the new app in the user code, the function should be called to confirm the operability of the app or vice versa about its non-operability. If the app is working, then it is marked as valid. Otherwise, it is marked as not valid and rolls back to the previous working app. A reboot is performed, and the app is booted before the software update. Note: If during the first boot a new app the power goes out or the WDT works, then roll back will happen. Rollback is possible only between the apps with the same security versions.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK

Enable app anti-rollback support

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE

This option prevents rollback to previous firmware/application image with lower security version.

Default value:

- No (disabled) if *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE*

CONFIG_BOOTLOADER_APP_SECURE_VERSION

eFuse secure version of app

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE > CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK

The secure version is the sequence number stored in the header of each firmware. The security version is set in the bootloader, version is recorded in the eFuse field as the number of set ones. The allocated number of bits in the efuse field for storing the security version is limited (see BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD option).

Bootloader: When bootloader selects an app to boot, an app is selected that has a security version greater or equal that recorded in eFuse field. The app is booted with a higher (or equal) secure version.

The security version is worth increasing if in previous versions there is a significant vulnerability and their use is not acceptable.

Your partition table should has a scheme with ota_0 + ota_1 (without factory).

Default value:

- 0 if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*

CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD

Size of the efuse secure version field

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE > CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK

The size of the efuse secure version field. Its length is limited to 32 bits for ESP32 and 16 bits for ESP32-S2. This determines how many times the security version can be increased.

Range:

- from 1 to 32 if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*
- from 1 to 16 if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*

Default value:

- 32 if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*
- 16 if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*

CONFIG_BOOTLOADER_EFUSE_SECURE_VERSION_EMULATE

Emulate operations with efuse secure version(only test)

Found in: Bootloader config > CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE > CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK

This option allows to emulate read/write operations with all eFuses and efuse secure version. It allows to test anti-rollback implemention without permanent write eFuse bits. There should be an entry in partition table with following details: *emul_efuse, data, efuse, , 0x2000*.

This option enables: EFUSE_VIRTUAL and EFUSE_VIRTUAL_KEEP_IN_FLASH.

Default value:

- No (disabled) if *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*

CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP

Skip image validation when exiting deep sleep

Found in: Bootloader config

This option disables the normal validation of an image coming out of deep sleep (checksums, SHA256, and signature). This is a trade-off between wakeup performance from deep sleep, and image integrity checks.

Only enable this if you know what you are doing. It should not be used in conjunction with using deep_sleep() entry and changing the active OTA partition as this would skip the validation upon first load of the new OTA partition.

It is possible to enable this option with Secure Boot if "allow insecure options" is enabled, however it's strongly recommended to NOT enable it as it may allow a Secure Boot bypass.

Default value:

- No (disabled) if (*CONFIG_SECURE_BOOT* && *CONFIG_SECURE_BOOT_INSECURE*) || *CONFIG_SECURE_BOOT*

CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON

Skip image validation from power on reset (READ HELP FIRST)

Found in: Bootloader config

Some applications need to boot very quickly from power on. By default, the entire app binary is read from flash and verified which takes up a significant portion of the boot time.

Enabling this option will skip validation of the app when the SoC boots from power on. Note that in this case it's not possible for the bootloader to detect if an app image is corrupted in the flash, therefore it's not possible to safely fall back to a different app partition. Flash corruption of this kind is unlikely but can happen if there is a serious firmware bug or physical damage.

Following other reset types, the bootloader will still validate the app image. This increases the chances that flash corruption resulting in a crash can be detected following soft reset, and the bootloader will fall back to a valid app image. To increase the chances of successfully recovering from a flash corruption event, keep the option BOOTLOADER_WDT_ENABLE enabled and consider also enabling BOOTLOADER_WDT_DISABLE_IN_USER_CODE - then manually disable the RTC Watchdog once the app is running. In addition, enable both the Task and Interrupt watchdog timers with reset options set.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_SKIP_VALIDATE_ALWAYS

Skip image validation always (READ HELP FIRST)

Found in: Bootloader config

Selecting this option prevents the bootloader from ever validating the app image before booting it. Any flash corruption of the selected app partition will make the entire SoC unbootable.

Although flash corruption is a very rare case, it is not recommended to select this option. Consider selecting "Skip image validation from power on reset" instead. However, if boot time is the only important factor then it can be enabled.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC

Reserve RTC FAST memory for custom purposes

Found in: Bootloader config

This option allows the customer to place data in the RTC FAST memory, this area remains valid when rebooted, except for power loss. This memory is located at a fixed address and is available for both the bootloader and the application. (The application and bootoloader must be compiled with the same option). The RTC FAST memory has access only through PRO_CPU.

Default value:

- No (disabled)

CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC_IN_CRC

Include custom memory in the CRC calculation

Found in: Bootloader config > CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC

This option allows the customer to use the legacy bootloader behavior when the RTC FAST memory CRC calculation takes place. When this option is enabled, the allocated user custom data will be taken into account in the CRC calculcation. This means that any change to the custom data would need a CRC update to prevent the bootloader from marking this data as corrupted. If this option is disabled, the custom data will not be taken into account when calculating the RTC FAST memory CRC. The user custom data can be changed freely, without the need to update the CRC. THIS OPTION MUST BE THE SAME FOR BOTH THE BOOTLOADER AND THE APPLICATION BUILDS.

Default value:

- No (disabled) if *CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC*

CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC_SIZE

Size in bytes for custom purposes

Found in: Bootloader config > CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC

This option reserves in RTC FAST memory the area for custom purposes. If you want to create your own bootloader and save more information in this area of memory, you can increase it. It must be a multiple of 4 bytes. This area (rtc_retain_mem_t) is reserved and has access from the bootloader and an application.

Default value:

- 0 if *CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC*

Security features

Contains:

- *CONFIG_SECURE_BOOT_INSECURE*
- *CONFIG_SECURE_SIGNED_APPS_SCHEME*
- *CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT*
- *CONFIG_SECURE_FLASH_CHECK_ENC_EN_IN_APP*
- *CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_SIZE*
- *CONFIG_SECURE_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE*
- *CONFIG_SECURE_FLASH_ENC_ENABLED*
- *CONFIG_SECURE_BOOT*
- *CONFIG_SECURE_FLASH_ENCRYPT_ONLY_IMAGE_LEN_IN_APP_PART*
- *CONFIG_SECURE_BOOTLOADER_KEY_ENCODING*
- *Potentially insecure options*
- *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*
- *CONFIG_SECURE_BOOT_VERIFICATION_KEY*

- *CONFIG_SECURE_BOOTLOADER_MODE*
- *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES*
- *CONFIG_SECURE_UART_ROM_DL_MODE*
- *CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT*

CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT

Require signed app images

Found in: Security features

Require apps to be signed to verify their integrity.

This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement.

CONFIG_SECURE_SIGNED_APPS_SCHEME

App Signing Scheme

Found in: Security features

Select the Secure App signing scheme. Depends on the Chip Revision. There are two secure boot versions:

1. **Secure boot V1**
   - Legacy custom secure boot scheme. Supported in ESP32 SoC.
2. **Secure boot V2**
   - RSA based secure boot scheme. Supported in ESP32-ECO3 (ESP32 Chip Revision 3 onwards), ESP32-S2, ESP32-C3, ESP32-S3 SoCs.
   - ECDSA based secure boot scheme. Supported in ESP32-C2 SoC.

Available options:

- ECDSA (SECURE_SIGNED_APPS_ECDSA_SCHEME)
  Embeds the ECDSA public key in the bootloader and signs the application with an ECDSA key. Refer to the documentation before enabling.
- RSA (SECURE_SIGNED_APPS_RSA_SCHEME)
  Appends the RSA-3072 based Signature block to the application. Refer to <Secure Boot Version 2 documentation link> before enabling.
- ECDSA (V2) (SECURE_SIGNED_APPS_ECDSA_V2_SCHEME)
  For Secure boot V2 (e.g., ESP32-C2 SoC), appends ECDSA based signature block to the application. Refer to documentation before enabling.

CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_SIZE

ECDSA key size

Found in: Security features

Select the ECDSA key size. Two key sizes are supported

- 192 bit key using NISTP192 curve
- 256 bit key using NISTP256 curve (Recommended)

The advantage of using 256 bit key is the extra randomness which makes it difficult to be bruteforced compared to 192 bit key. At present, both key sizes are practically implausible to bruteforce.

Available options:

- Using ECC curve NISTP192 (SECURE_BOOT_ECDSA_KEY_LEN_192_BITS)
- Using ECC curve NISTP256 (Recommended) (SECURE_BOOT_ECDSA_KEY_LEN_256_BITS)

CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT

Bootloader verifies app signatures

Found in: Security features

If this option is set, the bootloader will be compiled with code to verify that an app is signed before booting it.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

Default value:

- No (disabled) if *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT* && SECURE_SIGNED_APPS_ECDSA_SCHEME

CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT

Verify app signature on update

Found in: Security features

If this option is set, any OTA updated apps will have the signature verified before being considered valid.

When enabled, the signature is automatically checked whenever the esp_ota_ops.h APIs are used for OTA updates, or esp_image_format.h APIs are used to verify apps.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.

Default value:

- Yes (enabled) if *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*

CONFIG_SECURE_BOOT

Enable hardware Secure Boot in bootloader (READ DOCS FIRST)

Found in: Security features

Build a bootloader which enables Secure Boot on first boot.

Once enabled, Secure Boot will not boot a modified bootloader. The bootloader will only load a partition table or boot an app if the data has a verified digital signature. There are implications for reflashing updated apps once secure boot is enabled.

When enabling secure boot, JTAG and ROM BASIC Interpreter are permanently disabled by default.

Default value:

- No (disabled)

CONFIG_SECURE_BOOT_VERSION

Select secure boot version

Found in: Security features > CONFIG_SECURE_BOOT

Select the Secure Boot Version. Depends on the Chip Revision. Secure Boot V2 is the new RSA / ECDSA based secure boot scheme.

- RSA based scheme is supported in ESP32 (Revision 3 onwards), ESP32-S2, ESP32-C3 (ECO3), ESP32-S3.
- ECDSA based scheme is supported in ESP32-C2 SoC.

Please note that, RSA or ECDSA secure boot is property of specific SoC based on its HW design, supported crypto accelerators, die-size, cost and similar parameters. Please note that RSA scheme has requirement for bigger key sizes but at the same time it is comparatively faster than ECDSA verification.

Secure Boot V1 is the AES based (custom) secure boot scheme supported in ESP32 SoC.

Available options:

- Enable Secure Boot version 1 (SECURE_BOOT_V1_ENABLED)
  Build a bootloader which enables secure boot version 1 on first boot. Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.
- Enable Secure Boot version 2 (SECURE_BOOT_V2_ENABLED)
  Build a bootloader which enables Secure Boot version 2 on first boot. Refer to Secure Boot V2 section of the ESP-IDF Programmer's Guide for this version before enabling.

CONFIG_SECURE_BOOTLOADER_MODE

Secure bootloader mode

Found in: Security features

Available options:

- One-time flash (SECURE_BOOTLOADER_ONE_TIME_FLASH)
  On first boot, the bootloader will generate a key which is not readable externally or by software. A digest is generated from the bootloader image itself. This digest will be verified on each subsequent boot.
  Enabling this option means that the bootloader cannot be changed after the first time it is booted.
- Reflashable (SECURE_BOOTLOADER_REFLASHABLE)
  Generate a reusable secure bootloader key, derived (via SHA-256) from the secure boot signing key.
  This allows the secure bootloader to be re-flashed by anyone with access to the secure boot signing key.
  This option is less secure than one-time flash, because a leak of the digest key from one device allows reflashing of any device that uses it.

CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES

Sign binaries during build

Found in: Security features

Once secure boot or signed app requirement is enabled, app images are required to be signed.

If enabled (default), these binary files are signed as part of the build process. The file named in "Secure boot private signing key" will be used to sign the image.

If disabled, unsigned app/partition data will be built. They must be signed manually using espsecure.py. Version 1 to enable ECDSA Based Secure Boot and Version 2 to enable RSA based Secure Boot. (for example, on a remote signing server.)

CONFIG_SECURE_BOOT_SIGNING_KEY

Secure boot private signing key

Found in: Security features > CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES

Path to the key file used to sign app images.

Key file is an ECDSA private key (NIST256p curve) in PEM format for Secure Boot V1. Key file is an RSA private key in PEM format for Secure Boot V2.

Path is evaluated relative to the project directory.

You can generate a new signing key by running the following command: espsecure.py generate_signing_key secure_boot_signing_key.pem

See the Secure Boot section of the ESP-IDF Programmer's Guide for this version for details.

Default value:

- "secure_boot_signing_key.pem" if *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES*

CONFIG_SECURE_BOOT_VERIFICATION_KEY

Secure boot public signature verification key

Found in: Security features

Path to a public key file used to verify signed images. Secure Boot V1: This ECDSA public key is compiled into the bootloader and/or app, to verify app images.

Key file is in raw binary format, and can be extracted from a PEM formatted private key using the espsecure.py extract_public_key command.

Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.

CONFIG_SECURE_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE

Enable Aggressive key revoke strategy

Found in: Security features

If this option is set, ROM bootloader will revoke the public key digest burned in efuse block if it fails to verify the signature of software bootloader with it. Revocation of keys does not happen when enabling secure boot. Once secure boot is enabled, key revocation checks will be done on subsequent boot-up, while verifying the software bootloader

This feature provides a strong resistance against physical attacks on the device.

NOTE: Once a digest slot is revoked, it can never be used again to verify an image This can lead to permanent bricking of the device, in case all keys are revoked because of signature verification failure.

Default value:

- No (disabled) if *CONFIG_SECURE_BOOT* && SOC_SUPPORT_SECURE_BOOT_REVOKE_KEY

CONFIG_SECURE_BOOTLOADER_KEY_ENCODING

Hardware Key Encoding

Found in: Security features

In reflashable secure bootloader mode, a hardware key is derived from the signing key (with SHA-256) and can be written to eFuse with espefuse.py.

Normally this is a 256-bit key, but if 3/4 Coding Scheme is used on the device then the eFuse key is truncated to 192 bits.

This configuration item doesn't change any firmware code, it only changes the size of key binary which is generated at build time.

Available options:

- No encoding (256 bit key) (SECURE_BOOTLOADER_KEY_ENCODING_256BIT)
- 3/4 encoding (192 bit key) (SECURE_BOOTLOADER_KEY_ENCODING_192BIT)

CONFIG_SECURE_BOOT_INSECURE

Allow potentially insecure options

Found in: Security features

You can disable some of the default protections offered by secure boot, in order to enable testing or a custom combination of security features.

Only enable these options if you are very sure.

Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.

Default value:

- No (disabled) if *CONFIG_SECURE_BOOT*

CONFIG_SECURE_FLASH_ENC_ENABLED

Enable flash encryption on boot (READ DOCS FIRST)

Found in: Security features

If this option is set, flash contents will be encrypted by the bootloader on first boot.

Note: After first boot, the system will be permanently encrypted. Re-flashing an encrypted system is complicated and not always possible.

Read *Flash Encryption* before enabling.

Default value:

- No (disabled)

CONFIG_SECURE_FLASH_ENCRYPTION_KEYSIZE

Size of generated AES-XTS key

Found in: Security features > CONFIG_SECURE_FLASH_ENC_ENABLED

Size of generated AES-XTS key.

- AES-128 uses a 256-bit key (32 bytes) derived from 128 bits (16 bytes) burned in half Efuse key block. Internally, it calculates SHA256(128 bits)
- AES-128 uses a 256-bit key (32 bytes) which occupies one Efuse key block.
- AES-256 uses a 512-bit key (64 bytes) which occupies two Efuse key blocks.

This setting is ignored if either type of key is already burned to Efuse before the first boot. In this case, the pre-burned key is used and no new key is generated.

Available options:

- AES-128 key derived from 128 bits (SHA256(128 bits)) (SECURE_FLASH_ENCRYPTION_AES128_DERIVED)
- AES-128 (256-bit key) (SECURE_FLASH_ENCRYPTION_AES128)
- AES-256 (512-bit key) (SECURE_FLASH_ENCRYPTION_AES256)

CONFIG_SECURE_FLASH_ENCRYPTION_MODE

Enable usage mode

Found in: Security features > CONFIG_SECURE_FLASH_ENC_ENABLED

By default Development mode is enabled which allows ROM download mode to perform flash encryption operations (plaintext is sent to the device, and it encrypts it internally and writes ciphertext to flash.) This mode is not secure, it's possible for an attacker to write their own chosen plaintext to flash.

Release mode should always be selected for production or manufacturing. Once enabled it's no longer possible for the device in ROM Download Mode to use the flash encryption hardware.

Refer to the Flash Encryption section of the ESP-IDF Programmer's Guide for details.

Available options:

- Development (NOT SECURE) (SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT)
- Release (SECURE_FLASH_ENCRYPTION_MODE_RELEASE)

Potentially insecure options Contains:

- *CONFIG_SECURE_BOOT_V2_ALLOW_EFUSE_RD_DIS*
- *CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION*
- *CONFIG_SECURE_BOOT_ALLOW_JTAG*
- *CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC*
- *CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC*
- *CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC*
- *CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE*
- *CONFIG_SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS*
- *CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED*

CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC

Leave ROM BASIC Interpreter available on reset

Found in: Security features > Potentially insecure options

By default, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

When either flash encryption or secure boot are enabled, the default is to disable this BASIC fallback mode permanently via eFuse.

If this option is set, this eFuse is not burned and the BASIC ROM Console may remain accessible. Only set this option in testing environments.

Default value:

- No (disabled) if *CONFIG_SECURE_BOOT_INSECURE* || SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_BOOT_ALLOW_JTAG

Allow JTAG Debugging

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable JTAG (across entire chip) on first boot when either secure boot or flash encryption is enabled.

Setting this option leaves JTAG on for debugging, which negates all protections of flash encryption and some of the protections of secure boot.

Only set this option in testing environments.

Default value:

- No (disabled) if *CONFIG_SECURE_BOOT_INSECURE* || SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION

Allow app partition length not 64KB aligned

Found in: Security features > Potentially insecure options

If not set (default), app partition size must be a multiple of 64KB. App images are padded to 64KB length, and the bootloader checks any trailing bytes after the signature (before the next 64KB boundary) have not been written. This is because flash cache maps entire 64KB pages into the address space. This prevents an attacker from appending unverified data after the app image in the flash, causing it to be mapped into the address space.

Setting this option allows the app partition length to be unaligned, and disables padding of the app image to this length. It is generally not recommended to set this option, unless you have a legacy partitioning scheme which doesn't support 64KB aligned partition lengths.

CONFIG_SECURE_BOOT_V2_ALLOW_EFUSE_RD_DIS

Allow additional read protecting of efuses

Found in: Security features > Potentially insecure options

If not set (default, recommended), on first boot the bootloader will burn the WR_DIS_RD_DIS efuse when Secure Boot is enabled. This prevents any more efuses from being read protected.

If this option is set, it will remain possible to write the EFUSE_RD_DIS efuse field after Secure Boot is enabled. This may allow an attacker to read-protect the BLK2 efuse (for ESP32) and BLOCK4-BLOCK10 (i.e. BLOCK_KEY0-BLOCK_KEY5)(for other chips) holding the public key digest, causing an immediate denial of service and possibly allowing an additional fault injection attack to bypass the signature protection.

NOTE: Once a BLOCK is read-protected, the application will read all zeros from that block

NOTE: If "UART ROM download mode (Permanently disabled (recommended))" or "UART ROM download mode (Permanently switch to Secure mode (recommended))" is set, then it is __NOT__ possible to read/write efuses using espefuse.py utility. However, efuse can be read/written from the application

CONFIG_SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS

Leave unused digest slots available (not revoke)

Found in: Security features > Potentially insecure options

If not set (default), during startup in the app all unused digest slots will be revoked. To revoke unused slot will be called esp_efuse_set_digest_revoke(num_digest) for each digest. Revoking unused digest slots makes ensures that no trusted keys can be added later by an attacker. If set, it means that you have a plan to use unused digests slots later.

Default value:

- No (disabled) if *CONFIG_SECURE_BOOT_INSECURE* && SOC_EFUSE_REVOKE_BOOT_KEY_DIGESTS

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC

Leave UART bootloader encryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader encryption access on first boot. If set, the UART bootloader will still be able to access hardware encryption.

It is recommended to only set this option in testing environments.

Default value:

- No (disabled) if SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC

Leave UART bootloader decryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader decryption access on first boot. If set, the UART bootloader will still be able to access hardware decryption.

Only set this option in testing environments. Setting this option allows complete bypass of flash encryption.

Default value:

- No (disabled) if SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE

Leave UART bootloader flash cache enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader flash cache access on first boot. If set, the UART bootloader will still be able to access the flash cache.

Only set this option in testing environments.

Default value:

- No (disabled) if SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED

Require flash encryption to be already enabled

Found in: Security features > Potentially insecure options

If not set (default), and flash encryption is not yet enabled in eFuses, the 2nd stage bootloader will enable flash encryption: generate the flash encryption key and program eFuses. If this option is set, and flash encryption is not yet enabled, the bootloader will error out and reboot. If flash encryption is enabled in eFuses, this option does not change the bootloader behavior.

Only use this option in testing environments, to avoid accidentally enabling flash encryption on the wrong device. The device needs to have flash encryption already enabled using espefuse.py.

Default value:

- No (disabled) if SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT

CONFIG_SECURE_FLASH_ENCRYPT_ONLY_IMAGE_LEN_IN_APP_PART

Encrypt only the app image that is present in the partition of type app

Found in: Security features

If set, optimise encryption time for the partition of type APP, by only encrypting the app image that is present in the partition, instead of the whole partition. The image length used for encryption is derived from the image metadata, which includes the size of the app image, checksum, hash and also the signature sector when secure boot is enabled.

If not set (default), the whole partition of type APP would be encrypted, which increases the encryption time but might be useful if there is any custom data appended to the firmware image.

Default value:

- No (disabled) if *CONFIG_SECURE_FLASH_ENC_ENABLED* && *CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED*

CONFIG_SECURE_FLASH_CHECK_ENC_EN_IN_APP

Check Flash Encryption enabled on app startup

Found in: Security features

If set (default), in an app during startup code, there is a check of the flash encryption eFuse bit is on (as the bootloader should already have set it). The app requires this bit is on to continue work otherwise abort.

If not set, the app does not care if the flash encryption eFuse bit is set or not.

Default value:

- Yes (enabled) if *CONFIG_SECURE_FLASH_ENC_ENABLED*

CONFIG_SECURE_UART_ROM_DL_MODE

UART ROM download mode

Found in: Security features

Available options:

- UART ROM download mode (Permanently disabled (recommended)) (SECURE_DISABLE_ROM_DL_MODE)
  If set, during startup the app will burn an eFuse bit to permanently disable the UART ROM Download Mode. This prevents any future use of esptool.py, espefuse.py and similar tools.
  Once disabled, if the SoC is booted with strapping pins set for ROM Download Mode then an error is printed instead.
  It is recommended to enable this option in any production application where Flash Encryption and/or Secure Boot is enabled and access to Download Mode is not required.
  It is also possible to permanently disable Download Mode by calling esp_efuse_disable_rom_download_mode() at runtime.
- UART ROM download mode (Permanently switch to Secure mode (recommended)) (SECURE_ENABLE_SECURE_ROM_DL_MODE)
  If set, during startup the app will burn an eFuse bit to permanently switch the UART ROM Download Mode into a separate Secure Download mode. This option can only work if Download Mode is not already disabled by eFuse.
  Secure Download mode limits the use of Download Mode functions to update SPI config, changing baud rate, basic flash write and a command to return a summary of currently enabled security features (*get_security_info*).
  Secure Download mode is not compatible with the esptool.py flasher stub feature, espefuse.py, read/writing memory or registers, encrypted download, or any other features that interact with unsupported Download Mode commands.
  Secure Download mode should be enabled in any application where Flash Encryption and/or Secure Boot is enabled. Disabling this option does not immediately cancel the benefits of the security features, but it increases the potential "attack surface" for an attacker to try and bypass them with a successful physical attack.
  It is also possible to enable secure download mode at runtime by calling esp_efuse_enable_rom_secure_download_mode()
  Note: Secure Download mode is not available for ESP32 (includes revisions till ECO3).
- UART ROM download mode (Enabled (not recommended)) (SECURE_INSECURE_ALLOW_DL_MODE)
  This is a potentially insecure option. Enabling this option will allow the full UART download mode to stay enabled. This option SHOULD NOT BE ENABLED for production use cases.

Application manager

Contains:

- *CONFIG_APP_EXCLUDE_PROJECT_NAME_VAR*
- *CONFIG_APP_EXCLUDE_PROJECT_VER_VAR*
- *CONFIG_APP_PROJECT_VER_FROM_CONFIG*
- *CONFIG_APP_RETRIEVE_LEN_ELF_SHA*
- *CONFIG_APP_COMPILE_TIME_DATE*

CONFIG_APP_COMPILE_TIME_DATE

Use time/date stamp for app

Found in: Application manager

If set, then the app will be built with the current time/date stamp. It is stored in the app description structure. If not set, time/date stamp will be excluded from app image. This can be useful for getting the same binary image files made from the same source, but at different times.

Default value:

- Yes (enabled)

CONFIG_APP_EXCLUDE_PROJECT_VER_VAR

Exclude PROJECT_VER from firmware image

Found in: Application manager

The PROJECT_VER variable from the build system will not affect the firmware image. This value will not be contained in the esp_app_desc structure.

Default value:

- No (disabled)

CONFIG_APP_EXCLUDE_PROJECT_NAME_VAR

Exclude PROJECT_NAME from firmware image

Found in: Application manager

The PROJECT_NAME variable from the build system will not affect the firmware image. This value will not be contained in the esp_app_desc structure.

Default value:

- No (disabled)

CONFIG_APP_PROJECT_VER_FROM_CONFIG

Get the project version from Kconfig

Found in: Application manager

If this is enabled, then config item APP_PROJECT_VER will be used for the variable PROJECT_VER. Other ways to set PROJECT_VER will be ignored.

Default value:

- No (disabled)

CONFIG_APP_PROJECT_VER

Project version

Found in: Application manager > CONFIG_APP_PROJECT_VER_FROM_CONFIG

Project version

Default value:

- 1 if *CONFIG_APP_PROJECT_VER_FROM_CONFIG*

CONFIG_APP_RETRIEVE_LEN_ELF_SHA

The length of APP ELF SHA is stored in RAM(chars)

Found in: Application manager

At startup, the app will read this many hex characters from the embedded APP ELF SHA-256 hash value and store it in static RAM. This ensures the app ELF SHA-256 value is always available if it needs

to be printed by the panic handler code. Changing this value will change the size of a static buffer, in bytes.

Range:
- from 8 to 64

Default value:
- 16

Serial flasher config

Contains:

- *CONFIG_ESPTOOLPY_AFTER*
- *CONFIG_ESPTOOLPY_BEFORE*
- *CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE*
- *CONFIG_ESPTOOLPY_NO_STUB*
- *CONFIG_ESPTOOLPY_FLASH_SAMPLE_MODE*
- *CONFIG_ESPTOOLPY_FLASHSIZE*
- *CONFIG_ESPTOOLPY_FLASHMODE*
- *CONFIG_ESPTOOLPY_FLASHFREQ*

CONFIG_ESPTOOLPY_NO_STUB

Disable download stub

Found in: Serial flasher config

The flasher tool sends a precompiled download stub first by default. That stub allows things like compressed downloads and more. Usually you should not need to disable that feature

Default value:
- No (disabled)

CONFIG_ESPTOOLPY_FLASHMODE

Flash SPI mode

Found in: Serial flasher config

Mode the flash chip is flashed in, as well as the default mode for the binary to run in.

Available options:
- QIO (ESPTOOLPY_FLASHMODE_QIO)
- QOUT (ESPTOOLPY_FLASHMODE_QOUT)
- DIO (ESPTOOLPY_FLASHMODE_DIO)
- DOUT (ESPTOOLPY_FLASHMODE_DOUT)
- OPI (ESPTOOLPY_FLASHMODE_OPI)

CONFIG_ESPTOOLPY_FLASH_SAMPLE_MODE

Flash Sampling Mode

Found in: Serial flasher config

Available options:
- STR Mode (ESPTOOLPY_FLASH_SAMPLE_MODE_STR)
- DTR Mode (ESPTOOLPY_FLASH_SAMPLE_MODE_DTR)

CONFIG_ESPTOOLPY_FLASHFREQ

Flash SPI speed

Found in: Serial flasher config

Available options:

- 120 MHz (READ DOCS FIRST) (ESPTOOLPY_FLASHFREQ_120M)
  - Optional feature for QSPI Flash. Read docs and enable *CONFIG_SPI_FLASH_HPM_ENA* first!
- 80 MHz (ESPTOOLPY_FLASHFREQ_80M)
- 60 MHz (ESPTOOLPY_FLASHFREQ_60M)
- 48 MHz (ESPTOOLPY_FLASHFREQ_48M)
- 40 MHz (ESPTOOLPY_FLASHFREQ_40M)
- 30 MHz (ESPTOOLPY_FLASHFREQ_30M)
- 26 MHz (ESPTOOLPY_FLASHFREQ_26M)
- 24 MHz (ESPTOOLPY_FLASHFREQ_24M)
- 20 MHz (ESPTOOLPY_FLASHFREQ_20M)
- 15 MHz (ESPTOOLPY_FLASHFREQ_15M)

CONFIG_ESPTOOLPY_FLASHSIZE

Flash size

Found in: Serial flasher config

SPI flash size, in megabytes

Available options:

- 1 MB (ESPTOOLPY_FLASHSIZE_1MB)
- 2 MB (ESPTOOLPY_FLASHSIZE_2MB)
- 4 MB (ESPTOOLPY_FLASHSIZE_4MB)
- 8 MB (ESPTOOLPY_FLASHSIZE_8MB)
- 16 MB (ESPTOOLPY_FLASHSIZE_16MB)
- 32 MB (ESPTOOLPY_FLASHSIZE_32MB)
- 64 MB (ESPTOOLPY_FLASHSIZE_64MB)
- 128 MB (ESPTOOLPY_FLASHSIZE_128MB)

CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE

Detect flash size when flashing bootloader

Found in: Serial flasher config

If this option is set, flashing the project will automatically detect the flash size of the target chip and update the bootloader image before it is flashed.

Enabling this option turns off the image protection against corruption by a SHA256 digest. Updating the bootloader image before flashing would invalidate the digest.

Default value:

- No (disabled)

CONFIG_ESPTOOLPY_BEFORE

Before flashing

Found in: Serial flasher config

Configure whether esptool.py should reset the ESP32 before flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- Reset to bootloader (ESPTOOLPY_BEFORE_RESET)
- No reset (ESPTOOLPY_BEFORE_NORESET)

CONFIG_ESPTOOLPY_AFTER

After flashing

Found in: Serial flasher config

Configure whether esptool.py should reset the ESP32 after flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- Reset after flashing (ESPTOOLPY_AFTER_RESET)
- Stay in bootloader (ESPTOOLPY_AFTER_NORESET)

Partition Table

Contains:

- *CONFIG_PARTITION_TABLE_CUSTOM_FILENAME*
- *CONFIG_PARTITION_TABLE_MD5*
- *CONFIG_PARTITION_TABLE_OFFSET*
- *CONFIG_PARTITION_TABLE_TYPE*

CONFIG_PARTITION_TABLE_TYPE

Partition Table

Found in: Partition Table

The partition table to flash to the ESP32. The partition table determines where apps, data and other resources are expected to be found.

The predefined partition table CSV descriptions can be found in the components/partition_table directory. These are mostly intended for example and development use, it's expect that for production use you will copy one of these CSV files and create a custom partition CSV for your application.

Available options:

- Single factory app, no OTA (PARTITION_TABLE_SINGLE_APP)
  This is the default partition table, designed to fit into a 2MB or larger flash with a single 1MB app partition.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp.csv
  This partition table is not suitable for an app that needs OTA (over the air update) capability.
- Single factory app (large), no OTA (PARTITION_TABLE_SINGLE_APP_LARGE)
  This is a variation of the default partition table, that expands the 1MB app partition size to 1.5MB to fit more code.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_large.csv
  This partition table is not suitable for an app that needs OTA (over the air update) capability.
- Factory app, two OTA definitions (PARTITION_TABLE_TWO_OTA)
  This is a basic OTA-enabled partition table with a factory app partition plus two OTA app partitions. All are 1MB, so this partition table requires 4MB or larger flash size.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_two_ota.csv

- Custom partition table CSV (PARTITION_TABLE_CUSTOM)
  Specify the path to the partition table CSV to use for your project.
  Consult the Partition Table section in the ESP-IDF Programmers Guide for more information.
- Single factory app, no OTA, encrypted NVS (PARTITION_TABLE_SINGLE_APP_ENCRYPTED_NVS)
  This is a variation of the default "Single factory app, no OTA" partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_encr_nvs.csv
- Single factory app (large), no OTA, encrypted NVS (PARTITION_TABLE_SINGLE_APP_LARGE_ENC_NVS)
  This is a variation of the "Single factory app (large), no OTA" partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_large_encr_nvs.csv
- Factory app, two OTA definitions, encrypted NVS (PARTITION_TABLE_TWO_OTA_ENCRYPTED_NVS)
  This is a variation of the "Factory app, two OTA definitions" partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_two_ota_encr_nvs.csv

CONFIG_PARTITION_TABLE_CUSTOM_FILENAME

Custom partition CSV file

Found in: Partition Table

Name of the custom partition CSV filename. This path is evaluated relative to the project root directory.

Default value:

- "partitions.csv"

CONFIG_PARTITION_TABLE_OFFSET

Offset of partition table

Found in: Partition Table

The address of partition table (by default 0x8000). Allows you to move the partition table, it gives more space for the bootloader. Note that the bootloader and app will both need to be compiled with the same PARTITION_TABLE_OFFSET value.

This number should be a multiple of 0x1000.

Note that partition offsets in the partition table CSV file may need to be changed if this value is set to a higher value. To have each partition offset adapt to the configured partition table offset, leave all partition offsets blank in the CSV file.

Default value:

- "0x8000"

CONFIG_PARTITION_TABLE_MD5

Generate an MD5 checksum for the partition table

Found in: Partition Table

Generate an MD5 checksum for the partition table for protecting the integrity of the table. The generation should be turned off for legacy bootloaders which cannot recognize the MD5 checksum in the partition table.

Default value:

- Yes (enabled) if *CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS*

Compiler options

Contains:

- *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*
- *CONFIG_COMPILER_FLOAT_LIB_FROM*
- *CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT*
- *CONFIG_COMPILER_DUMP_RTL_FILES*
- *CONFIG_COMPILER_WARN_WRITE_STRINGS*
- *CONFIG_COMPILER_CXX_EXCEPTIONS*
- *CONFIG_COMPILER_CXX_RTTI*
- *CONFIG_COMPILER_OPTIMIZATION*
- *CONFIG_COMPILER_HIDE_PATHS_MACROS*
- *CONFIG_COMPILER_STACK_CHECK_MODE*

CONFIG_COMPILER_OPTIMIZATION

Optimization Level

Found in: Compiler options

This option sets compiler optimization level (gcc -O argument) for the app.

- The "Default" setting will add the -0g flag to CFLAGS.
- The "Size" setting will add the -0s flag to CFLAGS.
- The "Performance" setting will add the -O2 flag to CFLAGS.
- The "None" setting will add the -O0 flag to CFLAGS.

The "Size" setting cause the compiled code to be smaller and faster, but may lead to difficulties of correlating code addresses to source file lines when debugging.

The "Performance" setting causes the compiled code to be larger and faster, but will be easier to correlated code addresses to source file lines.

"None" with -O0 produces compiled code without optimization.

Note that custom optimization levels may be unsupported.

Compiler optimization for the IDF bootloader is set separately, see the BOOTLOADER_COMPILER_OPTIMIZATION setting.

Available options:

- Debug (-Og) (COMPILER_OPTIMIZATION_DEFAULT)
- Optimize for size (-Os) (COMPILER_OPTIMIZATION_SIZE)
- Optimize for performance (-O2) (COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (COMPILER_OPTIMIZATION_NONE)

CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL

Assertion level

Found in: Compiler options

Assertions can be:

- Enabled. Failure will print verbose assertion details. This is the default.

- Set to "silent" to save code size (failed assertions will abort() but user needs to use the aborting address to find the line number with the failed assertion.)
- Disabled entirely (not recommended for most configurations.) -DNDEBUG is added to CPPFLAGS in this case.

Available options:

- Enabled (COMPILER_OPTIMIZATION_ASSERTIONS_ENABLE)
  Enable assertions. Assertion content and line number will be printed on failure.
- Silent (saves code size) (COMPILER_OPTIMIZATION_ASSERTIONS_SILENT)
  Enable silent assertions. Failed assertions will abort(), user needs to use the aborting address to find the line number with the failed assertion.
- Disabled (sets -DNDEBUG) (COMPILER_OPTIMIZATION_ASSERTIONS_DISABLE)
  If assertions are disabled, -DNDEBUG is added to CPPFLAGS.

CONFIG_COMPILER_FLOAT_LIB_FROM

Compiler float lib source

Found in: Compiler options

In the soft-fp part of libgcc, riscv version is written in C, and handles all edge cases in IEEE754, which makes it larger and performance is slow.

RVfplib is an optimized RISC-V library for FP arithmetic on 32-bit integer processors, for single and double-precision FP. RVfplib is "fast", but it has a few exceptions from IEEE 754 compliance.

Available options:

- libgcc (COMPILER_FLOAT_LIB_FROM_GCCLIB)
- librvfp (COMPILER_FLOAT_LIB_FROM_RVFPLIB)

CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT

Disable messages in ESP_RETURN_ON_* and ESP_EXIT_ON_* macros

Found in: Compiler options

If enabled, the error messages will be discarded in following check macros: - ESP_RETURN_ON_ERROR - ESP_EXIT_ON_ERROR - ESP_RETURN_ON_FALSE - ESP_EXIT_ON_FALSE

Default value:

- No (disabled)

CONFIG_COMPILER_HIDE_PATHS_MACROS

Replace ESP-IDF and project paths in binaries

Found in: Compiler options

When expanding the __FILE__ and __BASE_FILE__ macros, replace paths inside ESP-IDF with paths relative to the placeholder string "IDF", and convert paths inside the project directory to relative paths.

This allows building the project with assertions or other code that embeds file paths, without the binary containing the exact path to the IDF or project directories.

This option passes -fmacro-prefix-map options to the GCC command line. To replace additional paths in your binaries, modify the project CMakeLists.txt file to pass custom -fmacro-prefix-map or -ffile-prefix-map arguments.

Default value:

- Yes (enabled)

CONFIG_COMPILER_CXX_EXCEPTIONS

Enable C++ exceptions

Found in: Compiler options

Enabling this option compiles all IDF C++ files with exception support enabled.

Disabling this option disables C++ exception support in all compiled files, and any libstdc++ code which throws an exception will abort instead.

Enabling this option currently adds an additional ~500 bytes of heap overhead when an exception is thrown in user code for the first time.

Default value:

- No (disabled)

Contains:

- *CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE*

CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE

Emergency Pool Size

Found in: Compiler options > CONFIG_COMPILER_CXX_EXCEPTIONS

Size (in bytes) of the emergency memory pool for C++ exceptions. This pool will be used to allocate memory for thrown exceptions when there is not enough memory on the heap.

Default value:

- 0 if *CONFIG_COMPILER_CXX_EXCEPTIONS*

CONFIG_COMPILER_CXX_RTTI

Enable C++ run-time type info (RTTI)

Found in: Compiler options

Enabling this option compiles all C++ files with RTTI support enabled. This increases binary size (typically by tens of kB) but allows using dynamic_cast conversion and typeid operator.

Default value:

- No (disabled)

CONFIG_COMPILER_STACK_CHECK_MODE

Stack smashing protection mode

Found in: Compiler options

Stack smashing protection mode. Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, program is halted. Protection has the following modes:

- In NORMAL mode (GCC flag: -fstack-protector) only functions that call alloca, and functions with buffers larger than 8 bytes are protected.
- STRONG mode (GCC flag: -fstack-protector-strong) is like NORMAL, but includes additional functions to be protected –those that have local array definitions, or have references to local frame addresses.
- In OVERALL mode (GCC flag: -fstack-protector-all) all functions are protected.

Modes have the following impact on code performance and coverage:

- performance: NORMAL > STRONG > OVERALL
- coverage: NORMAL < STRONG < OVERALL

The performance impact includes increasing the amount of stack memory required for each task.

Available options:

- None (COMPILER_STACK_CHECK_MODE_NONE)
- Normal (COMPILER_STACK_CHECK_MODE_NORM)
- Strong (COMPILER_STACK_CHECK_MODE_STRONG)
- Overall (COMPILER_STACK_CHECK_MODE_ALL)

CONFIG_COMPILER_WARN_WRITE_STRINGS

Enable -Wwrite-strings warning flag

Found in: Compiler options

Adds -Wwrite-strings flag for the C/C++ compilers.

For C, this gives string constants the type `const char[]` so that copying the address of one into a non-const `char \*` pointer produces a warning. This warning helps to find at compile time code that tries to write into a string constant.

For C++, this warns about the deprecated conversion from string literals to `char \*`.

Default value:

- No (disabled)

CONFIG_COMPILER_DUMP_RTL_FILES

Dump RTL files during compilation

Found in: Compiler options

If enabled, RTL files will be produced during compilation. These files can be used by other tools, for example to calculate call graphs.

Component config

Contains:

- *ADC and ADC Calibration*
- *Application Level Tracing*
- *Bluetooth*
- *Common ESP-related*
- *Core dump*
- *Driver Configurations*
- *eFuse Bit Manager*
- *CONFIG_BLE_MESH*
- *ESP HTTP client*
- *ESP HTTPS OTA*
- *ESP HTTPS server*
- *ESP NETIF Adapter*
- *ESP PSRAM*
- *ESP Ringbuf*
- *ESP System Settings*
- *ESP-MQTT Configurations*
- *ESP-TLS*
- *Ethernet*
- *Event Loop Library*
- *FAT Filesystem support*
- *FreeRTOS*
- *GDB Stub*
- *Hardware Abstraction Layer (HAL) and Low Level (LL)*

- *Hardware Settings*
- *Heap memory debugging*
- *High resolution timer (esp_timer)*
- *HTTP Server*
- *IPC (Inter-Processor Call)*
- *LCD and Touch Panel*
- *Log output*
- *LWIP*
- *Main Flash configuration*
- *mbedTLS*
- *Newlib*
- *NVS*
- *OpenThread*
- *PHY*
- *Power Management*
- *Protocomm*
- *PThreads*
- *SoC Settings*
- *SPI Flash driver*
- *SPIFFS Configuration*
- *Supplicant*
- *TCP Transport*
- *Ultra Low Power (ULP) Co-processor*
- *Unity unit testing library*
- *Virtual file system*
- *Wear Levelling*
- *Wi-Fi*
- *Wi-Fi Provisioning Manager*

Application Level Tracing Contains:

- *CONFIG_APPTRACE_DESTINATION1*
- *CONFIG_APPTRACE_DESTINATION2*
- *FreeRTOS SystemView Tracing*
- *CONFIG_APPTRACE_GCOV_ENABLE*
- *CONFIG_APPTRACE_BUF_SIZE*
- *CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX*
- *CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH*
- *CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO*
- *CONFIG_APPTRACE_UART_BAUDRATE*
- *CONFIG_APPTRACE_UART_RX_GPIO*
- *CONFIG_APPTRACE_UART_RX_BUFF_SIZE*
- *CONFIG_APPTRACE_UART_TASK_PRIO*
- *CONFIG_APPTRACE_UART_TX_MSG_SIZE*
- *CONFIG_APPTRACE_UART_TX_GPIO*
- *CONFIG_APPTRACE_UART_TX_BUFF_SIZE*

CONFIG_APPTRACE_DESTINATION1

Data Destination 1

Found in: Component config > Application Level Tracing

Select destination for application trace: JTAG or none (to disable).

Available options:

- JTAG (APPTRACE_DEST_JTAG)
- None (APPTRACE_DEST_NONE)

CONFIG_APPTRACE_DESTINATION2

Data Destination 2

Found in: Component config > Application Level Tracing

Select destination for application trace: UART(XX) or none (to disable).

Available options:

- UART0 (APPTRACE_DEST_UART0)
- UART1 (APPTRACE_DEST_UART1)
- UART2 (APPTRACE_DEST_UART2)
- USB_CDC (APPTRACE_DEST_USB_CDC)
- None (APPTRACE_DEST_UART_NONE)

CONFIG_APPTRACE_UART_TX_GPIO

UART TX on GPIO#

Found in: Component config > Application Level Tracing

This GPIO is used for UART TX pin.

CONFIG_APPTRACE_UART_RX_GPIO

UART RX on GPIO#

Found in: Component config > Application Level Tracing

This GPIO is used for UART RX pin.

CONFIG_APPTRACE_UART_BAUDRATE

UART baud rate

Found in: Component config > Application Level Tracing

This baud rate is used for UART.

The app’s maximum baud rate depends on the UART clock source. If Power Management is disabled, the UART clock source is the APB clock and all baud rates in the available range will be sufficiently accurate. If Power Management is enabled, REF_TICK clock source is used so the baud rate is divided from 1MHz. Baud rates above 1Mbps are not possible and values between 500Kbps and 1Mbps may not be accurate.

CONFIG_APPTRACE_UART_RX_BUFF_SIZE

UART RX ring buffer size

Found in: Component config > Application Level Tracing

Size of the UART input ring buffer. This size related to the baudrate, system tick frequency and amount of data to transfer. The data placed to this buffer before sent out to the interface.

CONFIG_APPTRACE_UART_TX_BUFF_SIZE

UART TX ring buffer size

Found in: Component config > Application Level Tracing

Size of the UART output ring buffer. This size related to the baudrate, system tick frequency and amount of data to transfer.

CONFIG_APPTRACE_UART_TX_MSG_SIZE

UART TX message size

Found in: Component config > Application Level Tracing

Maximum size of the single message to transfer.

CONFIG_APPTRACE_UART_TASK_PRIO

UART Task Priority

Found in: Component config > Application Level Tracing

UART task priority. In case of high events rate, this parameter could be changed up to (config-MAX_PRIORITIES-1).

Range:
- from 1 to 32

Default value:
- 1

CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO

Timeout for flushing last trace data to host on panic

Found in: Component config > Application Level Tracing

Timeout for flushing last trace data to host in case of panic. In ms. Use -1 to disable timeout and wait forever.

CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH

Threshold for flushing last trace data to host on panic

Found in: Component config > Application Level Tracing

Threshold for flushing last trace data to host on panic in post-mortem mode. This is minimal amount of data needed to perform flush. In bytes.

CONFIG_APPTRACE_BUF_SIZE

Size of the apptrace buffer

Found in: Component config > Application Level Tracing

Size of the memory buffer for trace data in bytes.

CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX

Size of the pending data buffer

Found in: Component config > Application Level Tracing

Size of the buffer for events in bytes. It is useful for buffering events from the time critical code (scheduler, ISRs etc). If this parameter is 0 then events will be discarded when main HW buffer is full.

FreeRTOS SystemView Tracing Contains:

- *CONFIG_APPTRACE_SV_CPU*
- *CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE*
- *CONFIG_APPTRACE_SV_MAX_TASKS*

- *CONFIG_APPTRACE_SV_EVT_IDLE_ENABLE*
- *CONFIG_APPTRACE_SV_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_START_READY_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_STOP_READY_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TASK_TERMINATE_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE*
- *CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE*
- *CONFIG_APPTRACE_SV_TS_SOURCE*
- *CONFIG_APPTRACE_SV_EVT_OVERFLOW_ENABLE*
- *CONFIG_APPTRACE_SV_BUF_WAIT_TMO*

CONFIG_APPTRACE_SV_ENABLE

SystemView Tracing Enable

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables supporrt for SEGGER SystemView tracing functionality.

CONFIG_APPTRACE_SV_DEST

SystemView destination

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_APPTRACE_SV_ENABLE

SystemView witt transfer data trough defined interface.

Available options:

- Data destination JTAG (APPTRACE_SV_DEST_JTAG)
  Send SEGGER SystemView events through JTAG interface.
- Data destination UART (APPTRACE_SV_DEST_UART)
  Send SEGGER SystemView events through UART interface.

CONFIG_APPTRACE_SV_CPU

CPU to trace

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Define the CPU to trace by SystemView.

Available options:

- CPU0 (APPTRACE_SV_DEST_CPU_0)
  Send SEGGER SystemView events for Pro CPU.
- CPU1 (APPTRACE_SV_DEST_CPU_1)
  Send SEGGER SystemView events for App CPU.

CONFIG_APPTRACE_SV_TS_SOURCE

Timer to use as timestamp source

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

SystemView needs to use a hardware timer as the source of timestamps when tracing. This option selects the timer for it.

Available options:

- CPU cycle counter (CCOUNT) (APPTRACE_SV_TS_SOURCE_CCOUNT)
- General Purpose Timer (Timer Group) (APPTRACE_SV_TS_SOURCE_GPTIMER)

- esp_timer high resolution timer (APPTRACE_SV_TS_SOURCE_ESP_TIMER)

CONFIG_APPTRACE_SV_MAX_TASKS

Maximum supported tasks

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Configures maximum supported tasks in sysview debug

CONFIG_APPTRACE_SV_BUF_WAIT_TMO

Trace buffer wait timeout

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Configures timeout (in us) to wait for free space in trace buffer. Set to -1 to wait forever and avoid lost events.

CONFIG_APPTRACE_SV_EVT_OVERFLOW_ENABLE

Trace Buffer Overflow Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Trace Buffer Overflow" event.

CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE

ISR Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "ISR Enter" event.

CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE

ISR Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "ISR Exit" event.

CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE

ISR Exit to Scheduler Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "ISR to Scheduler" event.

CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABLE

Task Start Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Start Execution" event.

CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE

Task Stop Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Stop Execution" event.

CONFIG_APPTRACE_SV_EVT_TASK_START_READY_ENABLE

Task Start Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Start Ready State" event.

CONFIG_APPTRACE_SV_EVT_TASK_STOP_READY_ENABLE

Task Stop Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Stop Ready State" event.

CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE

Task Create Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Create" event.

CONFIG_APPTRACE_SV_EVT_TASK_TERMINATE_ENABLE

Task Terminate Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Task Terminate" event.

CONFIG_APPTRACE_SV_EVT_IDLE_ENABLE

System Idle Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "System Idle" event.

CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE

Timer Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Timer Enter" event.

CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE

Timer Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables "Timer Exit" event.

CONFIG_APPTRACE_GCOV_ENABLE

GCOV to Host Enable

Found in: Component config > Application Level Tracing

Enables support for GCOV data transfer to host.

CONFIG_APPTRACE_GCOV_DUMP_TASK_STACK_SIZE

Gcov dump task stack size

Found in: Component config > Application Level Tracing > CONFIG_APPTRACE_GCOV_ENABLE

Configures stack size of Gcov dump task

Default value:

- 2048 if *CONFIG_APPTRACE_GCOV_ENABLE*

Bluetooth Contains:

- *Bluedroid Options*
- *CONFIG_BT_ENABLED*
- *Controller Options*
- *CONFIG_BT_HCI_LOG_DEBUG_EN*
- *NimBLE Options*
- *CONFIG_BT_RELEASE_IRAM*

CONFIG_BT_ENABLED

Bluetooth

Found in: Component config > Bluetooth

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.

CONFIG_BT_HOST

Host

Found in: Component config > Bluetooth > CONFIG_BT_ENABLED

This helps to choose Bluetooth host stack

Available options:

- Bluedroid - Dual-mode (BT_BLUEDROID_ENABLED)
  This option is recommended for classic Bluetooth or for dual-mode usecases
- NimBLE - BLE only (BT_NIMBLE_ENABLED)
  This option is recommended for BLE only usecases to save on memory
- Disabled (BT_CONTROLLER_ONLY)
  This option is recommended when you want to communicate directly with the controller (without any host) or when you are using any other host stack not supported by Espressif (not mentioned here).

CONFIG_BT_CONTROLLER

Controller

Found in: Component config > Bluetooth > CONFIG_BT_ENABLED

This helps to choose Bluetooth controller stack

Available options:

- Enabled (BT_CONTROLLER_ENABLED)
  This option is recommended for Bluetooth controller usecases
- Disabled (BT_CONTROLLER_DISABLED)
  This option is recommended for Bluetooth Host only usecases

Bluedroid Options Contains:

- *CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK*
- *CONFIG_BT_BLUEDROID_MEM_DEBUG*
- *CONFIG_BT_BTU_TASK_STACK_SIZE*
- *CONFIG_BT_BTC_TASK_STACK_SIZE*
- *CONFIG_BT_BLE_ENABLED*
- *BT DEBUG LOG LEVEL*
- *CONFIG_BT_ACL_CONNECTIONS*
- *CONFIG_BT_SMP_MAX_BONDS*
- *CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST*
- *CONFIG_BT_CLASSIC_ENABLED*
- *CONFIG_BT_HID_ENABLED*
- *CONFIG_BT_STACK_NO_LOG*
- *CONFIG_BT_BLE_42_FEATURES_SUPPORTED*
- *CONFIG_BT_BLE_50_FEATURES_SUPPORTED*
- *CONFIG_BT_BLE_HIGH_DUTY_ADV_INTERVAL*
- *CONFIG_BT_MULTI_CONNECTION_ENBALE*
- *CONFIG_BT_BLE_FEAT_PERIODIC_ADV_SYNC_TRANSFER*
- *CONFIG_BT_BLE_FEAT_CREATE_SYNC_ENH*
- *CONFIG_BT_BLE_FEAT_PERIODIC_ADV_ENH*
- *CONFIG_BT_MAX_DEVICE_NAME_LEN*
- *CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN*
- *CONFIG_BT_SSP_ENABLED*
- *CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE*
- *CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT*
- *CONFIG_BT_BLE_RPA_TIMEOUT*
- *CONFIG_BT_BLE_RPA_SUPPORTED*
- *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY*
- *CONFIG_BT_HFP_WBS_ENABLE*

CONFIG_BT_BTC_TASK_STACK_SIZE

Bluetooth event (callback to application) task stack size

Found in: Component config > Bluetooth > Bluedroid Options

This select btc task stack size

Default value:

- 3072 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE

The cpu core which Bluedroid run

Found in: Component config > Bluetooth > Bluedroid Options

Which the cpu core to run Bluedroid. Can choose core0 and core1. Can not specify no-affinity.

Available options:

- Core 0 (PRO CPU) (BT_BLUEDROID_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (BT_BLUEDROID_PINNED_TO_CORE_1)

CONFIG_BT_BTU_TASK_STACK_SIZE

Bluetooth Bluedroid Host Stack task stack size

Found in: Component config > Bluetooth > Bluedroid Options

This select btu task stack size

Default value:

- 4352 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLUEDROID_MEM_DEBUG

Bluedroid memory debug

Found in: Component config > Bluetooth > Bluedroid Options

Bluedroid memory debug

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_CLASSIC_ENABLED

Classic Bluetooth

Found in: Component config > Bluetooth > Bluedroid Options

For now this option needs "SMP_ENABLE" to be set to yes

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_CLASSIC_BQB_ENABLED

Host Qualitifcation support for Classic Bluetooth

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED

This enables functionalities of Host qualification for Classic Bluetooth.

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_A2DP_ENABLE

A2DP

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED

Advanced Audio Distrubution Profile

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_SPP_ENABLED

SPP

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED

This enables the Serial Port Profile

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_L2CAP_ENABLED

BT L2CAP

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED

This enables the Logical Link Control and Adaptation Layer Protocol. Only supported classic bluetooth.

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_HFP_ENABLE

Hands Free/Handset Profile

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED

Hands Free Unit and Audio Gateway can be included simultaneously but they cannot run simultaneously due to internal limitations.

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

Contains:

- *CONFIG_BT_HFP_AG_ENABLE*
- *CONFIG_BT_HFP_AUDIO_DATA_PATH*
- *CONFIG_BT_HFP_CLIENT_ENABLE*

CONFIG_BT_HFP_CLIENT_ENABLE

Hands Free Unit

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED > CONFIG_BT_HFP_ENABLE

Default value:

- Yes (enabled) if *CONFIG_BT_HFP_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_HFP_AG_ENABLE

Audio Gateway

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED > CONFIG_BT_HFP_ENABLE

Default value:

- Yes (enabled) if *CONFIG_BT_HFP_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_HFP_AUDIO_DATA_PATH

audio(SCO) data path

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED > CONFIG_BT_HFP_ENABLE

SCO data path, i.e. HCI or PCM. This option is set using API "esp_bredr_sco_datapath_set" in Bluetooth host. Default SCO data path can also be set in Bluetooth Controller.

Available options:

- PCM (BT_HFP_AUDIO_DATA_PATH_PCM)
- HCI (BT_HFP_AUDIO_DATA_PATH_HCI)

CONFIG_BT_HFP_WBS_ENABLE

Wide Band Speech

Found in: Component config > Bluetooth > Bluedroid Options

This enables Wide Band Speech. Should disable it when SCO data path is PCM. Otherwise there will be no data transmited via GPIOs.

Default value:

- Yes (enabled) if BT_HFP_AUDIO_DATA_PATH_HCI && BT_BLUEDROID_ENABLED

CONFIG_BT_HID_ENABLED

Classic BT HID

Found in: Component config > Bluetooth > Bluedroid Options

This enables the BT HID Host

Default value:

- No (disabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

Contains:

- *CONFIG_BT_HID_DEVICE_ENABLED*
- *CONFIG_BT_HID_HOST_ENABLED*

CONFIG_BT_HID_HOST_ENABLED

Classic BT HID Host

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_HID_ENABLED

This enables the BT HID Host

Default value:

- No (disabled) if *CONFIG_BT_HID_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_HID_DEVICE_ENABLED

Classic BT HID Device

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_HID_ENABLED

This enables the BT HID Device

CONFIG_BT_SSP_ENABLED

Secure Simple Pairing

Found in: Component config > Bluetooth > Bluedroid Options

This enables the Secure Simple Pairing. If disable this option, Bluedroid will only support Legacy Pairing

Default value:

- Yes (enabled) if *CONFIG_BT_CLASSIC_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_ENABLED

Bluetooth Low Energy

Found in: Component config > Bluetooth > Bluedroid Options

This enables Bluetooth Low Energy

Default value:

- Yes (enabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTS_ENABLE

Include GATT server module(GATTS)

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED

This option can be disabled when the app work only on gatt client mode

Default value:

- Yes (enabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTS_PPCP_CHAR_GAP

Enable Peripheral Preferred Connection Parameters characteristic in GAP service

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

This enables "Peripheral Preferred Connection Parameters" characteristic (UUID: 0x2A04) in GAP service that has connection parameters like min/max connection interval, slave latency and supervision timeout multiplier

Default value:

- No (disabled) if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_BLUFI_ENABLE

Include blufi function

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

This option can be close when the app does not require blufi function.

Default value:

- No (disabled) if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATT_MAX_SR_PROFILES

Max GATT Server Profiles

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

Maximum GATT Server Profiles Count

Range:

- from 1 to 32 if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

Default value:

- 8 if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_GATT_MAX_SR_ATTRIBUTES

Max GATT Service Attributes

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

Maximum GATT Service Attributes Count

Range:

- from 1 to 500 if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

Default value:

- 100 if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTS_SEND_SERVICE_CHANGE_MODE

GATTS Service Change Mode

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

Service change indication mode for GATT Server.

Available options:

- GATTS manually send service change indication (BT_GATTS_SEND_SERVICE_CHANGE_MANUAL)
  Manually send service change indication through API esp_ble_gatts_send_service_change_indication()
- GATTS automatically send service change indication (BT_GATTS_SEND_SERVICE_CHANGE_AUTO)
  Let Bluedroid handle the service change indication internally

CONFIG_BT_GATTS_ROBUST_CACHING_ENABLED

Enable Robust Caching on Server Side

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

This option enables the GATT robust caching feature on the server. if turned on, the Client Supported Features characteristic, Database Hash characteristic, and Server Supported Features characteristic will be included in the GAP SERVICE.

Default value:

- No (disabled) if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTS_DEVICE_NAME_WRITABLE

Allow to write device name by GATT clients

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

Enabling this option allows remote GATT clients to write device name

Default value:

- No (disabled) if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTS_APPEARANCE_WRITABLE

Allow to write appearance by GATT clients

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE

Enabling this option allows remote GATT clients to write appearance

Default value:

- No (disabled) if *CONFIG_BT_GATTS_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_ENABLE

Include GATT client module(GATTC)

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED

This option can be close when the app work only on gatt server mode

Default value:

- Yes (enabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_MAX_CACHE_CHAR

Max gattc cache characteristic for discover

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

Maximum GATTC cache characteristic count

Range:

- from 1 to 500 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

Default value:

- 40 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_NOTIF_REG_MAX

Max gattc notify(indication) register number

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

Maximum GATTC notify(indication) register number

Range:

- from 1 to 64 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

Default value:

- 5 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_CACHE_NVS_FLASH

Save gattc cache data to nvs flash

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

This select can save gattc cache data to nvs flash

Default value:

- No (disabled) if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_CONNECT_RETRY_COUNT

The number of attempts to reconnect if the connection establishment failed

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

The number of attempts to reconnect if the connection establishment failed

Range:

- from 0 to 255 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

Default value:

- 3 if *CONFIG_BT_GATTC_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_SMP_ENABLE

Include BLE security module(SMP)

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED

This option can be close when the app not used the ble security connect.

Default value:

- Yes (enabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_SMP_SLAVE_CON_PARAMS_UPD_ENABLE

Slave enable connection parameters update during pairing

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_BLE_SMP_ENABLE

In order to reduce the pairing time, slave actively initiates connection parameters update during pairing.

Default value:

- No (disabled) if *CONFIG_BT_BLE_SMP_ENABLE* && BT_BLUEDROID_ENABLED

CONFIG_BT_STACK_NO_LOG

Disable BT debug logs (minimize bin size)

Found in: Component config > Bluetooth > Bluedroid Options

This select can save the rodata code size

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

BT DEBUG LOG LEVEL

Contains:

- *CONFIG_BT_LOG_A2D_TRACE_LEVEL*
- *CONFIG_BT_LOG_APPL_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVCT_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVDT_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVRC_TRACE_LEVEL*
- *CONFIG_BT_LOG_BLUFI_TRACE_LEVEL*
- *CONFIG_BT_LOG_BNEP_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTC_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTIF_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTM_TRACE_LEVEL*
- *CONFIG_BT_LOG_GAP_TRACE_LEVEL*
- *CONFIG_BT_LOG_GATT_TRACE_LEVEL*
- *CONFIG_BT_LOG_HCI_TRACE_LEVEL*
- *CONFIG_BT_LOG_HID_TRACE_LEVEL*
- *CONFIG_BT_LOG_L2CAP_TRACE_LEVEL*
- *CONFIG_BT_LOG_MCA_TRACE_LEVEL*
- *CONFIG_BT_LOG_OSI_TRACE_LEVEL*
- *CONFIG_BT_LOG_PAN_TRACE_LEVEL*
- *CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL*
- *CONFIG_BT_LOG_SDP_TRACE_LEVEL*
- *CONFIG_BT_LOG_SMP_TRACE_LEVEL*

CONFIG_BT_LOG_HCI_TRACE_LEVEL

HCI layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for HCI layer

Available options:

- NONE (BT_LOG_HCI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_HCI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_HCI_TRACE_LEVEL_WARNING)
- API (BT_LOG_HCI_TRACE_LEVEL_API)
- EVENT (BT_LOG_HCI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_HCI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_HCI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTM_TRACE_LEVEL

BTM layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for BTM layer

Available options:

- NONE (BT_LOG_BTM_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTM_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTM_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTM_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTM_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTM_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTM_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_L2CAP_TRACE_LEVEL

L2CAP layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for L2CAP layer

Available options:

- NONE (BT_LOG_L2CAP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_L2CAP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_L2CAP_TRACE_LEVEL_WARNING)
- API (BT_LOG_L2CAP_TRACE_LEVEL_API)
- EVENT (BT_LOG_L2CAP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_L2CAP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_L2CAP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL

RFCOMM layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for RFCOMM layer

Available options:

- NONE (BT_LOG_RFCOMM_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_RFCOMM_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_RFCOMM_TRACE_LEVEL_WARNING)
- API (BT_LOG_RFCOMM_TRACE_LEVEL_API)
- EVENT (BT_LOG_RFCOMM_TRACE_LEVEL_EVENT)

- DEBUG (BT_LOG_RFCOMM_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_RFCOMM_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_SDP_TRACE_LEVEL

SDP layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for SDP layer

Available options:

- NONE (BT_LOG_SDP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_SDP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_SDP_TRACE_LEVEL_WARNING)
- API (BT_LOG_SDP_TRACE_LEVEL_API)
- EVENT (BT_LOG_SDP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_SDP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_SDP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_GAP_TRACE_LEVEL

GAP layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for GAP layer

Available options:

- NONE (BT_LOG_GAP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_GAP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_GAP_TRACE_LEVEL_WARNING)
- API (BT_LOG_GAP_TRACE_LEVEL_API)
- EVENT (BT_LOG_GAP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_GAP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_GAP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BNEP_TRACE_LEVEL

BNEP layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for BNEP layer

Available options:

- NONE (BT_LOG_BNEP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BNEP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BNEP_TRACE_LEVEL_WARNING)
- API (BT_LOG_BNEP_TRACE_LEVEL_API)
- EVENT (BT_LOG_BNEP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BNEP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BNEP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_PAN_TRACE_LEVEL

PAN layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for PAN layer

Available options:

- NONE (BT_LOG_PAN_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_PAN_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_PAN_TRACE_LEVEL_WARNING)
- API (BT_LOG_PAN_TRACE_LEVEL_API)
- EVENT (BT_LOG_PAN_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_PAN_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_PAN_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_A2D_TRACE_LEVEL

A2D layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for A2D layer

Available options:

- NONE (BT_LOG_A2D_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_A2D_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_A2D_TRACE_LEVEL_WARNING)
- API (BT_LOG_A2D_TRACE_LEVEL_API)
- EVENT (BT_LOG_A2D_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_A2D_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_A2D_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVDT_TRACE_LEVEL

AVDT layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for AVDT layer

Available options:

- NONE (BT_LOG_AVDT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVDT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVDT_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVDT_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVDT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVDT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVDT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVCT_TRACE_LEVEL

AVCT layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for AVCT layer

Available options:

- NONE (BT_LOG_AVCT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVCT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVCT_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVCT_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVCT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVCT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVCT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVRC_TRACE_LEVEL

AVRC layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for AVRC layer

Available options:

- NONE (BT_LOG_AVRC_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVRC_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVRC_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVRC_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVRC_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVRC_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVRC_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_MCA_TRACE_LEVEL

MCA layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for MCA layer

Available options:

- NONE (BT_LOG_MCA_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_MCA_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_MCA_TRACE_LEVEL_WARNING)
- API (BT_LOG_MCA_TRACE_LEVEL_API)
- EVENT (BT_LOG_MCA_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_MCA_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_MCA_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_HID_TRACE_LEVEL

HID layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for HID layer

Available options:

- NONE (BT_LOG_HID_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_HID_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_HID_TRACE_LEVEL_WARNING)
- API (BT_LOG_HID_TRACE_LEVEL_API)
- EVENT (BT_LOG_HID_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_HID_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_HID_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_APPL_TRACE_LEVEL

APPL layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for APPL layer

Available options:

- NONE (BT_LOG_APPL_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_APPL_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_APPL_TRACE_LEVEL_WARNING)
- API (BT_LOG_APPL_TRACE_LEVEL_API)

- EVENT (BT_LOG_APPL_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_APPL_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_APPL_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_GATT_TRACE_LEVEL

GATT layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for GATT layer

Available options:

- NONE (BT_LOG_GATT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_GATT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_GATT_TRACE_LEVEL_WARNING)
- API (BT_LOG_GATT_TRACE_LEVEL_API)
- EVENT (BT_LOG_GATT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_GATT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_GATT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_SMP_TRACE_LEVEL

SMP layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for SMP layer

Available options:

- NONE (BT_LOG_SMP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_SMP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_SMP_TRACE_LEVEL_WARNING)
- API (BT_LOG_SMP_TRACE_LEVEL_API)
- EVENT (BT_LOG_SMP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_SMP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_SMP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTIF_TRACE_LEVEL

BTIF layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for BTIF layer

Available options:

- NONE (BT_LOG_BTIF_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTIF_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTIF_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTIF_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTIF_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTIF_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTIF_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTC_TRACE_LEVEL

BTC layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for BTC layer

Available options:

- NONE (BT_LOG_BTC_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTC_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTC_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTC_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTC_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTC_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTC_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_OSI_TRACE_LEVEL

OSI layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for OSI layer

Available options:

- NONE (BT_LOG_OSI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_OSI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_OSI_TRACE_LEVEL_WARNING)
- API (BT_LOG_OSI_TRACE_LEVEL_API)
- EVENT (BT_LOG_OSI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_OSI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_OSI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BLUFI_TRACE_LEVEL

BLUFI layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for BLUFI layer

Available options:

- NONE (BT_LOG_BLUFI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BLUFI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BLUFI_TRACE_LEVEL_WARNING)
- API (BT_LOG_BLUFI_TRACE_LEVEL_API)
- EVENT (BT_LOG_BLUFI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BLUFI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BLUFI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_ACL_CONNECTIONS

BT/BLE MAX ACL CONNECTIONS(1~9)

Found in: Component config > Bluetooth > Bluedroid Options

Maximum BT/BLE connection count. The ESP32-C3/S3 chip supports a maximum of 10 instances, including ADV, SCAN and connections. The ESP32-C3/S3 chip can connect up to 9 devices if ADV or SCAN uses only one. If ADV and SCAN are both used, The ESP32-C3/S3 chip is connected to a maximum of 8 devices. Because Bluetooth cannot reclaim used instances once ADV or SCAN is used.

Range:

- from 1 to 9 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

Default value:

- 4 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_MULTI_CONNECTION_ENBALE

Enable BLE multi-conections

Found in: Component config > Bluetooth > Bluedroid Options

Enable this option if there are multiple connections

Default value:

- Yes (enabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST

BT/BLE will first malloc the memory from the PSRAM

Found in: Component config > Bluetooth > Bluedroid Options

This select can save the internal RAM if there have the PSRAM

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY

Use dynamic memory allocation in BT/BLE stack

Found in: Component config > Bluetooth > Bluedroid Options

This select can make the allocation of memory will become more flexible

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK

BLE queue congestion check

Found in: Component config > Bluetooth > Bluedroid Options

When scanning and scan duplicate is not enabled, if there are a lot of adv packets around or application layer handling adv packets is slow, it will cause the controller memory to run out. if enabled, adv packets will be lost when host queue is congested.

Default value:

- No (disabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_SMP_MAX_BONDS

BT/BLE maximum bond device count

Found in: Component config > Bluetooth > Bluedroid Options

The number of security records for peer devices.

CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN

Report adv data and scan response individually when BLE active scan

Found in: Component config > Bluetooth > Bluedroid Options

Originally, when doing BLE active scan, Bluedroid will not report adv to application layer until receive scan response. This option is used to disable the behavior. When enable this option, Bluedroid will report adv data or scan response to application layer immediately.

Memory reserved at start of DRAM for Bluetooth stack

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED && *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT

Timeout of BLE connection establishment

Found in: Component config > Bluetooth > Bluedroid Options

Bluetooth Connection establishment maximum time, if connection time exceeds this value, the connection establishment fails, ESP_GATTC_OPEN_EVT or ESP_GATTS_OPEN_EVT is triggered.

Range:
- from 1 to 60 if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

Default value:
- 30 if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_MAX_DEVICE_NAME_LEN

length of bluetooth device name

Found in: Component config > Bluetooth > Bluedroid Options

Bluetooth Device name length shall be no larger than 248 octets, If the broadcast data cannot contain the complete device name, then only the shortname will be displayed, the rest parts that can ' t fit in will be truncated.

Range:
- from 32 to 248 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

Default value:
- 32 if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_RPA_SUPPORTED

Update RPA to Controller

Found in: Component config > Bluetooth > Bluedroid Options

This enables controller RPA list function. For ESP32, ESP32 only support network privacy mode. If this option is enabled, ESP32 will only accept advertising packets from peer devices that contain private address, HW will not receive the advertising packets contain identity address after IRK changed. If this option is disabled, address resolution will be performed in the host, so the functions that require controller to resolve address in the white list cannot be used. This option is disabled by default on ESP32, please enable or disable this option according to your own needs.

For other BLE chips, devices support network privacy mode and device privacy mode, users can switch the two modes according to their own needs. So this option is enabled by default.

Default value:
- No (disabled) if BT_BLUEDROID_ENABLED && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_RPA_TIMEOUT

Timeout of resolvable private address

Found in: Component config > Bluetooth > Bluedroid Options

This set RPA timeout of Controller and Host. Default is 900 s (15 minutes). Range is 1 s to 1 hour (3600 s).

Range:
- from 1 to 3600 if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

Default value:
- 900 if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_50_FEATURES_SUPPORTED

Enable BLE 5.0 features

Found in: Component config > Bluetooth > Bluedroid Options

Enabling this option activates BLE 5.0 features. This option is universally supported in chips that support BLE, except for ESP32.

Default value:

- Yes (enabled) if *CONFIG_BT_BLE_ENABLED* && SOC_ESP_NIMBLE_CONTROLLER && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_42_FEATURES_SUPPORTED

Enable BLE 4.2 features

Found in: Component config > Bluetooth > Bluedroid Options

This enables BLE 4.2 features.

Default value:

- No (disabled) if *CONFIG_BT_BLE_ENABLED* && SOC_ESP_NIMBLE_CONTROLLER && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_FEAT_PERIODIC_ADV_SYNC_TRANSFER

Enable BLE periodic advertising sync transfer feature

Found in: Component config > Bluetooth > Bluedroid Options

This enables BLE periodic advertising sync transfer feature

Default value:

- No (disabled) if *CONFIG_BT_BLE_50_FEATURES_SUPPORTED* && SOC_ESP_NIMBLE_CONTROLLER && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_FEAT_PERIODIC_ADV_ENH

Enable periodic adv enhancements(adi support)

Found in: Component config > Bluetooth > Bluedroid Options

Enable the periodic advertising enhancements

Default value:

- No (disabled) if *CONFIG_BT_BLE_50_FEATURES_SUPPORTED* && SOC_ESP_NIMBLE_CONTROLLER && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_FEAT_CREATE_SYNC_ENH

Enable create sync enhancements(reporting disable and duplicate filtering enable support)

Found in: Component config > Bluetooth > Bluedroid Options

Enable the create sync enhancements

Default value:

- No (disabled) if *CONFIG_BT_BLE_50_FEATURES_SUPPORTED* && SOC_ESP_NIMBLE_CONTROLLER && BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_HIGH_DUTY_ADV_INTERVAL

Enable BLE high duty advertising interval feature

Found in: Component config > Bluetooth > Bluedroid Options

This enable BLE high duty advertising interval feature

Default value:

- No (disabled) if *CONFIG_BT_BLE_ENABLED* && BT_BLUEDROID_ENABLED

NimBLE Options Contains:

- *CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME*
- *CONFIG_BT_NIMBLE_HS_STOP_TIMEOUT_MS*
- *CONFIG_BT_NIMBLE_HOST_QUEUE_CONG_CHECK*
- *CONFIG_BT_NIMBLE_WHITELIST_SIZE*
- *CONFIG_BT_NIMBLE_BLE_GATT_BLOB_TRANSFER*
- *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT*
- *CONFIG_BT_NIMBLE_ROLE_BROADCASTER*
- *CONFIG_BT_NIMBLE_ROLE_CENTRAL*
- *CONFIG_BT_NIMBLE_HIGH_DUTY_ADV_ITVL*
- *CONFIG_BT_NIMBLE_MESH*
- *CONFIG_BT_NIMBLE_ROLE_OBSERVER*
- *CONFIG_BT_NIMBLE_ROLE_PERIPHERAL*
- *CONFIG_BT_NIMBLE_SECURITY_ENABLE*
- *CONFIG_BT_NIMBLE_BLUFI_ENABLE*
- *CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT*
- *CONFIG_BT_NIMBLE_USE_ESP_TIMER*
- *CONFIG_BT_NIMBLE_DEBUG*
- *CONFIG_BT_NIMBLE_HOST_BASED_PRIVACY*
- *CONFIG_BT_NIMBLE_HS_FLOW_CTRL*
- *CONFIG_BT_NIMBLE_VS_SUPPORT*
- *CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE*
- *CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN*
- *CONFIG_BT_NIMBLE_MAX_BONDS*
- *CONFIG_BT_NIMBLE_MAX_CCCDS*
- *CONFIG_BT_NIMBLE_MAX_CONNECTIONS*
- *CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM*
- *CONFIG_BT_NIMBLE_GATT_MAX_PROCS*
- *CONFIG_BT_NIMBLE_MEM_ALLOC_MODE*
- *Memory Settings*
- *CONFIG_BT_NIMBLE_LOG_LEVEL*
- *CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE*
- *CONFIG_BT_NIMBLE_CRYPTO_STACK_MBEDTLS*
- *CONFIG_BT_NIMBLE_NVS_PERSIST*
- *CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU*
- *CONFIG_BT_NIMBLE_RPA_TIMEOUT*
- *CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE*
- *CONFIG_BT_NIMBLE_TEST_THROUGHPUT_TEST*

CONFIG_BT_NIMBLE_MEM_ALLOC_MODE

Memory allocation strategy

Found in: Component config > Bluetooth > NimBLE Options

Allocation strategy for NimBLE host stack, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only

- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF
- Internal IRAM memory wherever applicable else internal DRAM

Available options:

- Internal memory (BT_NIMBLE_MEM_ALLOC_MODE_INTERNAL)
- External SPIRAM (BT_NIMBLE_MEM_ALLOC_MODE_EXTERNAL)
- Default alloc mode (BT_NIMBLE_MEM_ALLOC_MODE_DEFAULT)
- Internal IRAM (BT_NIMBLE_MEM_ALLOC_MODE_IRAM_8BIT)
  Allows to use IRAM memory region as 8bit accessible region.
  Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

CONFIG_BT_NIMBLE_LOG_LEVEL

NimBLE Host log verbosity

Found in: Component config > Bluetooth > NimBLE Options

Select NimBLE log level. Please make a note that the selected NimBLE log verbosity can not exceed the level set in "Component config –> Log output –> Default log verbosity".

Available options:

- No logs (BT_NIMBLE_LOG_LEVEL_NONE)
- Error logs (BT_NIMBLE_LOG_LEVEL_ERROR)
- Warning logs (BT_NIMBLE_LOG_LEVEL_WARNING)
- Info logs (BT_NIMBLE_LOG_LEVEL_INFO)
- Debug logs (BT_NIMBLE_LOG_LEVEL_DEBUG)

CONFIG_BT_NIMBLE_MAX_CONNECTIONS

Maximum number of concurrent connections

Found in: Component config > Bluetooth > NimBLE Options

Defines maximum number of concurrent BLE connections. For ESP32, user is expected to configure BTDM_CTRL_BLE_MAX_CONN from controller menu along with this option. Similarly for ESP32-C3 or ESP32-S3, user is expected to configure BT_CTRL_BLE_MAX_ACT from controller menu. For ESP32C2, ESP32C6 and ESP32H2, each connection will take about 1k DRAM.

Range:

- from 1 to 9 if SOC_ESP_NIMBLE_CONTROLLER && BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_BONDS

Maximum number of bonds to save across reboots

Found in: Component config > Bluetooth > NimBLE Options

Defines maximum number of bonds to save for peer security and our security

Default value:

- 3 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_CCCDS

Maximum number of CCC descriptors to save across reboots

Found in: Component config > Bluetooth > NimBLE Options

Defines maximum number of CCC descriptors to save

Default value:

- 8 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM

Maximum number of connection oriented channels

Found in: Component config > Bluetooth > NimBLE Options

Defines maximum number of BLE Connection Oriented Channels. When set to (0), BLE COC is not compiled in

Range:
- from 0 to 9 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Default value:
- 0 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE

The CPU core on which NimBLE host will run

Found in: Component config > Bluetooth > NimBLE Options

The CPU core on which NimBLE host will run. You can choose Core 0 or Core 1. Cannot specify no-affinity

Available options:
- Core 0 (PRO CPU) (BT_NIMBLE_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (BT_NIMBLE_PINNED_TO_CORE_1)

CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE

NimBLE Host task stack size

Found in: Component config > Bluetooth > NimBLE Options

This configures stack size of NimBLE host task

Default value:
- 5120 if *CONFIG_BLE_MESH* && BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED
- 4096 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ROLE_CENTRAL

Enable BLE Central role

Found in: Component config > Bluetooth > NimBLE Options

Enables central role

Default value:
- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ROLE_PERIPHERAL

Enable BLE Peripheral role

Found in: Component config > Bluetooth > NimBLE Options

Enable peripheral role

Default value:
- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ROLE_BROADCASTER

Enable BLE Broadcaster role

Found in: Component config > Bluetooth > NimBLE Options

Enables broadcaster role

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ROLE_OBSERVER

Enable BLE Observer role

Found in: Component config > Bluetooth > NimBLE Options

Enables observer role

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_NVS_PERSIST

Persist the BLE Bonding keys in NVS

Found in: Component config > Bluetooth > NimBLE Options

Enable this flag to make bonding persistent across device reboots

Default value:

- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SECURITY_ENABLE

Enable BLE SM feature

Found in: Component config > Bluetooth > NimBLE Options

Enable BLE sm feature

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Contains:

- *CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_ENCRYPTION*
- *CONFIG_BT_NIMBLE_SM_LEGACY*
- *CONFIG_BT_NIMBLE_SM_SC*

CONFIG_BT_NIMBLE_SM_LEGACY

Security manager legacy pairing

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE

Enable security manager legacy pairing

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_SECURITY_ENABLE* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SM_SC

Security manager secure connections (4.2)

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE

Enable security manager secure connections

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_SECURITY_ENABLE* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS

Use predefined public-private key pair

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE > CONFIG_BT_NIMBLE_SM_SC

If this option is enabled, SM uses predefined DH key pair as described in Core Specification, Vol. 3, Part H, 2.3.5.6.1. This allows to decrypt air traffic easily and thus should only be used for debugging.

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_SECURITY_ENABLE* && *CONFIG_BT_NIMBLE_SM_SC* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_ENCRYPTION

Enable LE encryption

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE

Enable encryption connection

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_SECURITY_ENABLE* && BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_DEBUG

Enable extra runtime asserts and host debugging

Found in: Component config > Bluetooth > NimBLE Options

This enables extra runtime asserts and host debugging

Default value:

- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME

BLE GAP default device name

Found in: Component config > Bluetooth > NimBLE Options

The Device Name characteristic shall contain the name of the device as an UTF-8 string. This name can be changed by using API ble_svc_gap_device_name_set()

Default value:

- “nimble” if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN

Maximum length of BLE device name in octets

Found in: Component config > Bluetooth > NimBLE Options

Device Name characteristic value shall be 0 to 248 octets in length

Default value:

- 31 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU

Preferred MTU size in octets

Found in: Component config > Bluetooth > NimBLE Options

This is the default value of ATT MTU indicated by the device during an ATT MTU exchange. This value can be changed using API ble_att_set_preferred_mtu()

Default value:

- 256 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE

External appearance of the device

Found in: Component config > Bluetooth > NimBLE Options

Standard BLE GAP Appearance value in HEX format e.g. 0x02C0

Default value:

- 0 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Memory Settings Contains:

- *CONFIG_BT_NIMBLE_ACL_BUF_COUNT*
- *CONFIG_BT_NIMBLE_ACL_BUF_SIZE*
- *CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE*
- *CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT*
- *CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT*
- *CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT*
- *CONFIG_BT_NIMBLE_MSYS_1_BLOCK_SIZE*
- *CONFIG_BT_NIMBLE_MSYS_2_BLOCK_COUNT*
- *CONFIG_BT_NIMBLE_MSYS_2_BLOCK_SIZE*

CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT

MSYS_1 Block Count

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

MSYS is a system level mbuf registry. For prepare write & prepare responses MBUFs are allocated out of msys_1 pool. For NIMBLE_MESH enabled cases, this block count is increased by 8 than user defined count.

Default value:

- 24 if SOC_ESP_NIMBLE_CONTROLLER && BT_NIMBLE_ENABLED
- 12 if SOC_ESP_NIMBLE_CONTROLLER && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MSYS_1_BLOCK_SIZE

MSYS_1 Block Size

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

Dynamic memory size of block 1

Default value:

- 128 if SOC_ESP_NIMBLE_CONTROLLER && BT_NIMBLE_ENABLED
- 256 if SOC_ESP_NIMBLE_CONTROLLER && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MSYS_2_BLOCK_COUNT

MSYS_2 Block Count

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

Dynamic memory count

Default value:

- 24 if BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MSYS_2_BLOCK_SIZE

MSYS_2 Block Size

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

Dynamic memory size of block 2

Default value:

- 320 if BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ACL_BUF_COUNT

ACL Buffer count

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

The number of ACL data buffers.

Default value:

- 24 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ACL_BUF_SIZE

ACL Buffer size

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

This is the maximum size of the data portion of HCI ACL data packets. It does not include the HCI data header (of 4 bytes)

Default value:

- 255 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE

HCI Event Buffer size

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

This is the size of each HCI event buffer in bytes. In case of extended advertising, packets can be fragmented. 257 bytes is the maximum size of a packet.

Default value:

- 257 if *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED
- 70 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT

High Priority HCI Event Buffer count

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

This is the high priority HCI events' buffer size. High-priority event buffers are for everything except advertising reports. If there are no free high-priority event buffers then host will try to allocate a low-priority buffer instead

Default value:

- 30 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT

Low Priority HCI Event Buffer count

Found in: Component config > Bluetooth > NimBLE Options > Memory Settings

This is the low priority HCI events' buffer size. Low-priority event buffers are only used for advertising reports. If there are no free low-priority event buffers, then an incoming advertising report will get dropped

Default value:

- 8 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_GATT_MAX_PROCS

Maximum number of GATT client procedures

Found in: Component config > Bluetooth > NimBLE Options

Maximum number of GATT client procedures that can be executed.

Default value:

- 4 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HS_FLOW_CTRL

Enable Host Flow control

Found in: Component config > Bluetooth > NimBLE Options

Enable Host Flow control

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED
- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_ITVL

Host Flow control interval

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_HS_FLOW_CTRL

Host flow control interval in msecs

Default value:

- 1000 if *CONFIG_BT_NIMBLE_HS_FLOW_CTRL* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_THRESH

Host Flow control threshold

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_HS_FLOW_CTRL

Host flow control threshold, if the number of free buffers are at or below this threshold, send an immediate number-of-completed-packets event

Default value:

- 2 if *CONFIG_BT_NIMBLE_HS_FLOW_CTRL* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT

Host Flow control on disconnect

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_HS_FLOW_CTRL

Enable this option to send number-of-completed-packets event to controller after disconnection

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_HS_FLOW_CTRL* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_RPA_TIMEOUT

RPA timeout in seconds

Found in: Component config > Bluetooth > NimBLE Options

Time interval between RPA address change.

Range:

- from 1 to 41400 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Default value:

- 900 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH

Enable BLE mesh functionality

Found in: Component config > Bluetooth > NimBLE Options

Enable BLE Mesh example present in upstream mynewt-nimble and not maintained by Espressif.

IDF maintains ESP-BLE-MESH as the official Mesh solution. Please refer to ESP-BLE-MESH guide at: *:doc:../esp32/api-guides/esp-ble-mesh/ble-mesh-index``*

Default value:

- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Contains:

- *CONFIG_BT_NIMBLE_MESH_PROVISIONER*
- *CONFIG_BT_NIMBLE_MESH_PROV*
- *CONFIG_BT_NIMBLE_MESH_GATT_PROXY*
- *CONFIG_BT_NIMBLE_MESH_FRIEND*
- *CONFIG_BT_NIMBLE_MESH_LOW_POWER*
- *CONFIG_BT_NIMBLE_MESH_PROXY*
- *CONFIG_BT_NIMBLE_MESH_RELAY*
- *CONFIG_BT_NIMBLE_MESH_DEVICE_NAME*
- *CONFIG_BT_NIMBLE_MESH_NODE_COUNT*

CONFIG_BT_NIMBLE_MESH_PROXY

Enable mesh proxy functionality

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable proxy. This is automatically set whenever NIMBLE_MESH_PB_GATT or NIMBLE_MESH_GATT_PROXY is set

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PROV

Enable BLE mesh provisioning

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable mesh provisioning

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PB_ADV

Enable mesh provisioning over advertising bearer

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH > CONFIG_BT_NIMBLE_MESH_PROV

Enable this option to allow the device to be provisioned over the advertising bearer

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_MESH_PROV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PB_GATT

Enable mesh provisioning over GATT bearer

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH > CONFIG_BT_NIMBLE_MESH_PROV

Enable this option to allow the device to be provisioned over the GATT bearer

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_MESH_PROV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_GATT_PROXY

Enable GATT Proxy functionality

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

This option enables support for the Mesh GATT Proxy Service, i.e. the ability to act as a proxy between a Mesh GATT Client and a Mesh network

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_RELAY

Enable mesh relay functionality

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Support for acting as a Mesh Relay Node

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_LOW_POWER

Enable mesh low power mode

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable this option to be able to act as a Low Power Node

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_FRIEND

Enable mesh friend functionality

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable this option to be able to act as a Friend Node

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_DEVICE_NAME

Set mesh device name

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

This value defines Bluetooth Mesh device/node name

Default value:

- “nimble-mesh-node” if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_NODE_COUNT

Set mesh node count

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Defines mesh node count.

Default value:

- 1 if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PROVISIONER

Enable BLE mesh provisioner

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable mesh provisioner.

Default value:

- 0 if *CONFIG_BT_NIMBLE_MESH* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_CRYPTO_STACK_MBEDTLS

Override TinyCrypt with mbedTLS for crypto computations

Found in: Component config > Bluetooth > NimBLE Options

Enable this option to choose mbedTLS instead of TinyCrypt for crypto computations.

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HS_STOP_TIMEOUT_MS

BLE host stop timeout in msec

Found in: Component config > Bluetooth > NimBLE Options

BLE Host stop procedure timeout in milliseconds.

Default value:

- 2000 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_HOST_BASED_PRIVACY

Enable host based privacy for random address.

Found in: Component config > Bluetooth > NimBLE Options

Use this option to do host based Random Private Address resolution. If this option is disabled then controller based privacy is used.

Default value:

- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT

Enable connection reattempts on connection establishment error

Found in: Component config > Bluetooth > NimBLE Options

Enable to make the NimBLE host to reattempt GAP connection on connection establishment failure.

Default value:

- No (disabled) if BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_CONN_REATTEMPT

Maximum number connection reattempts

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT

Defines maximum number of connection reattempts.

Range:

- from 1 to 255 if BT_NIMBLE_ENABLED && *CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT* && BT_NIMBLE_ENABLED

Default value:

- 3 if BT_NIMBLE_ENABLED && *CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable BLE 5 feature

Found in: Component config > Bluetooth > NimBLE Options

Enable BLE 5 feature

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED && (SOC_BLE_50_SUPPORTED || BT_CONTROLLER_ENABLED) && BT_NIMBLE_ENABLED

Contains:

- *CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_2M_PHY*
- *CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_CODED_PHY*
- *CONFIG_BT_NIMBLE_EXT_ADV*
- *CONFIG_BT_NIMBLE_BLE_POWER_CONTROL*
- *CONFIG_BT_NIMBLE_MAX_PERIODIC_SYNCS*

CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_2M_PHY

Enable 2M Phy

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable 2M-PHY

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_CODED_PHY

Enable coded Phy

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable coded-PHY

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_EXT_ADV

Enable extended advertising

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable this option to do extended advertising. Extended advertising will be supported from BLE 5.0 onwards.

Default value:

- No (disabled) if *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_EXT_ADV_INSTANCES

Maximum number of extended advertising instances.

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV

Change this option to set maximum number of extended advertising instances. Minimum there is always one instance of advertising. Enter how many more advertising instances you want. For ESP32C2, ESP32C6 and ESP32H2, each extended advertising instance will take about 0.5k DRAM.

Range:

- from 0 to 4 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

Default value:

- 1 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED
- 0 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_EXT_ADV_MAX_SIZE

Maximum length of the advertising data.

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV

Defines the length of the extended adv data. The value should not exceed 1650.

Range:

- from 0 to 1650 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

Default value:

- 1650 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED
- 0 if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV

Enable periodic advertisement.

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV

Enable this option to start periodic advertisement.

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_EXT_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_PERIODIC_ADV_SYNC_TRANSFER

Enable Transer Sync Events

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV > CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV

This enables controller transfer periodic sync events to host

Default value:

- Yes (enabled) if *CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV* && *CONFIG_BT_NIMBLE_EXT_ADV* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_PERIODIC_SYNCS

Maximum number of periodic advertising syncs

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Set this option to set the upper limit for number of periodic sync connections. This should be less than maximum connections allowed by controller.

Range:
- from 0 to 8 if *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED

Default value:
- 1 if *CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV* && *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED
- 0 if *CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT* && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_BLE_POWER_CONTROL

Enable support for BLE Power Control

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Set this option to enable the Power Control feature

CONFIG_BT_NIMBLE_WHITELIST_SIZE

BLE white list size

Found in: Component config > Bluetooth > NimBLE Options

BLE list size

Range:
- from 1 to 15 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

Default value:
- 12 if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_TEST_THROUGHPUT_TEST

Throughput Test Mode enable

Found in: Component config > Bluetooth > NimBLE Options

Enable the throughput test mode

Default value:
- No (disabled) if BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_BLUFI_ENABLE

Enable blufi functionality

Found in: Component config > Bluetooth > NimBLE Options

Set this option to enable blufi functionality.

Default value:
- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_USE_ESP_TIMER

Enable Esp Timer for Nimble

Found in: Component config > Bluetooth > NimBLE Options

Set this option to use Esp Timer which has higher priority timer instead of FreeRTOS timer

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_BLE_GATT_BLOB_TRANSFER

Blob transfer

Found in: Component config > Bluetooth > NimBLE Options

This option is used when data to be sent is more than 512 bytes. For peripheral role, BT_NIMBLE_MSYS_1_BLOCK_COUNT needs to be increased according to the need.

CONFIG_BT_NIMBLE_HIGH_DUTY_ADV_ITVL

Enable BLE high duty advertising interval feature

Found in: Component config > Bluetooth > NimBLE Options

This enable BLE high duty advertising interval feature

CONFIG_BT_NIMBLE_HOST_QUEUE_CONG_CHECK

BLE queue congestion check

Found in: Component config > Bluetooth > NimBLE Options

When scanning and scan duplicate is not enabled, if there are a lot of adv packets around or application layer handling adv packets is slow, it will cause the controller memory to run out. if enabled, adv packets will be lost when host queue is congested.

Default value:

- No (disabled) if BT_NIMBLE_ENABLED && BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_VS_SUPPORT

Enable support for VSC and VSE

Found in: Component config > Bluetooth > NimBLE Options

This option is used to enable support for sending Vendor Specific HCI commands and handling Vendor Specific HCI Events.

Controller Options Contains:

- *CONFIG_BTDM_CTRL_AUTO_LATENCY*
- *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP*
- *CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED*
- *CONFIG_BTDM_CTRL_BLE_MAX_CONN*
- *CONFIG_BTDM_BLE_SCAN_DUPL*
- *CONFIG_BTDM_BLE_SLEEP_CLOCK_ACCURACY*
- *CONFIG_BTDM_CTRL_MODE*
- *CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN*
- *CONFIG_BTDM_CTRL_BR_EDR_SCO_DATA_PATH*
- *CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN*
- *CONFIG_BTDM_CTRL_SCAN_BACKOFF_UPPERLIMITMAX*
- *CONFIG_BTDM_CTRL_HCI_MODE_CHOICE*

- *HCI UART(H4) Options*
- *CONFIG_BTDM_CTRL_HLI*
- *CONFIG_BTDM_CTRL_LEGACY_AUTH_VENDOR_EVT*
- *MODEM SLEEP Options*
- *CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG*
- *CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE*

CONFIG_BTDM_CTRL_MODE

Bluetooth controller mode (BR/EDR/BLE/DUALMODE)

Found in: Component config > Bluetooth > Controller Options

Specify the bluetooth controller mode (BR/EDR, BLE or dual mode).

Available options:
- BLE Only (BTDM_CTRL_MODE_BLE_ONLY)
- BR/EDR Only (BTDM_CTRL_MODE_BR_EDR_ONLY)
- Bluetooth Dual Mode (BTDM_CTRL_MODE_BTDM)

CONFIG_BTDM_CTRL_BLE_MAX_CONN

BLE Max Connections

Found in: Component config > Bluetooth > Controller Options

BLE maximum connections of bluetooth controller. Each connection uses 1KB static DRAM whenever the BT controller is enabled.

Range:
- from 1 to 9 if (BTDM_CTRL_MODE_BLE_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

Default value:
- 3 if (BTDM_CTRL_MODE_BLE_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN

BR/EDR ACL Max Connections

Found in: Component config > Bluetooth > Controller Options

BR/EDR ACL maximum connections of bluetooth controller. Each connection uses 1.2 KB DRAM whenever the BT controller is enabled.

Range:
- from 1 to 7 if (BTDM_CTRL_MODE_BR_EDR_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

Default value:
- 2 if (BTDM_CTRL_MODE_BR_EDR_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN

BR/EDR Sync(SCO/eSCO) Max Connections

Found in: Component config > Bluetooth > Controller Options

BR/EDR Synchronize maximum connections of bluetooth controller. Each connection uses 2 KB DRAM whenever the BT controller is enabled.

Range:

- from 0 to 3 if (BTDM_CTRL_MODE_BR_EDR_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

Default value:

- 0 if (BTDM_CTRL_MODE_BR_EDR_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_BR_EDR_SCO_DATA_PATH

BR/EDR Sync(SCO/eSCO) default data path

Found in: Component config > Bluetooth > Controller Options

SCO data path, i.e. HCI or PCM. SCO data can be sent/received through HCI synchronous packets, or the data can be routed to on-chip PCM module on ESP32. PCM input/output signals can be "matrixed" to GPIOs. The default data path can also be set using API "esp_bredr_sco_datapath_set"

Available options:

- HCI (BTDM_CTRL_BR_EDR_SCO_DATA_PATH_HCI)
- PCM (BTDM_CTRL_BR_EDR_SCO_DATA_PATH_PCM)

CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG

PCM Signal Config (Role and Polar)

Found in: Component config > Bluetooth > Controller Options

Default value:

- Yes (enabled) if BTDM_CTRL_BR_EDR_SCO_DATA_PATH_PCM && BT_CONTROLLER_ENABLED

Contains:

- *CONFIG_BTDM_CTRL_PCM_POLAR*
- *CONFIG_BTDM_CTRL_PCM_ROLE*

CONFIG_BTDM_CTRL_PCM_ROLE

PCM Role

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG

PCM role can be configured as PCM master or PCM slave

Available options:

- PCM Master (BTDM_CTRL_PCM_ROLE_MASTER)
- PCM Slave (BTDM_CTRL_PCM_ROLE_SLAVE)

CONFIG_BTDM_CTRL_PCM_POLAR

PCM Polar

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG

PCM polarity can be configured as Falling Edge or Rising Edge

Available options:

- Falling Edge (BTDM_CTRL_PCM_POLAR_FALLING_EDGE)
- Rising Edge (BTDM_CTRL_PCM_POLAR_RISING_EDGE)

CONFIG_BTDM_CTRL_AUTO_LATENCY

Auto latency

Found in: Component config > Bluetooth > Controller Options

BLE auto latency, used to enhance classic BT performance while classic BT and BLE are enabled at the same time.

Default value:

- No (disabled) if BTDM_CTRL_MODE_BTDM && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_LEGACY_AUTH_VENDOR_EVT

Legacy Authentication Vendor Specific Event Enable

Found in: Component config > Bluetooth > Controller Options

To protect from BIAS attack during Legacy authentication, Legacy authentication Vendor specific event should be enabled

Default value:

- Yes (enabled) if (BTDM_CTRL_MODE_BR_EDR_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE

The cpu core which bluetooth controller run

Found in: Component config > Bluetooth > Controller Options

Specify the cpu core to run bluetooth controller. Can not specify no-affinity.

Available options:

- Core 0 (PRO CPU) (BTDM_CTRL_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (BTDM_CTRL_PINNED_TO_CORE_1)

CONFIG_BTDM_CTRL_HCI_MODE_CHOICE

HCI mode

Found in: Component config > Bluetooth > Controller Options

Specify HCI mode as VHCI or UART(H4)

Available options:

- VHCI (BTDM_CTRL_HCI_MODE_VHCI)
  Normal option. Mostly, choose this VHCI when bluetooth host run on ESP32, too.
- UART(H4) (BTDM_CTRL_HCI_MODE_UART_H4)
  If use external bluetooth host which run on other hardware and use UART as the HCI interface, choose this option.

HCI UART(H4) Options

Contains:

- *CONFIG_BTDM_CTRL_HCI_UART_FLOW_CTRL_EN*
- *CONFIG_BTDM_CTRL_HCI_UART_BAUDRATE*
- *CONFIG_BTDM_CTRL_HCI_UART_NO*

CONFIG_BTDM_CTRL_HCI_UART_NO

UART Number for HCI

Found in: Component config > Bluetooth > Controller Options > HCI UART(H4) Options

Uart number for HCI. The available uart is UART1 and UART2.

Range:
- from 1 to 2 if BTDM_CTRL_HCI_MODE_UART_H4 && BT_CONTROLLER_ENABLED

Default value:
- 1 if BTDM_CTRL_HCI_MODE_UART_H4 && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_HCI_UART_BAUDRATE

UART Baudrate for HCI

Found in: Component config > Bluetooth > Controller Options > HCI UART(H4) Options

UART Baudrate for HCI. Please use standard baudrate.

Range:
- from 115200 to 921600 if BTDM_CTRL_HCI_MODE_UART_H4 && BT_CONTROLLER_ENABLED

Default value:
- 921600 if BTDM_CTRL_HCI_MODE_UART_H4 && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_HCI_UART_FLOW_CTRL_EN

Enable UART flow control

Found in: Component config > Bluetooth > Controller Options > HCI UART(H4) Options

Default value:
- Yes (enabled) if BTDM_CTRL_HCI_MODE_UART_H4 && BT_CONTROLLER_ENABLED

MODEM SLEEP Options

Contains:

- *CONFIG_BTDM_CTRL_LOW_POWER_CLOCK*
- *CONFIG_BTDM_CTRL_MODEM_SLEEP*

CONFIG_BTDM_CTRL_MODEM_SLEEP

Bluetooth modem sleep

Found in: Component config > Bluetooth > Controller Options > MODEM SLEEP Options

Enable/disable bluetooth controller low power mode.

Default value:
- Yes (enabled) if BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_MODEM_SLEEP_MODE

Bluetooth Modem sleep mode

Found in: Component config > Bluetooth > Controller Options > MODEM SLEEP Options > CONFIG_BTDM_CTRL_MODEM_SLEEP

To select which strategy to use for modem sleep

Available options:
- ORIG Mode(sleep with low power clock) (BTDM_CTRL_MODEM_SLEEP_MODE_ORIG)
  ORIG mode is a bluetooth sleep mode that can be used for dual mode controller. In this mode, bluetooth controller sleeps between BR/EDR frames and BLE events. A low power clock is used to maintain bluetooth reference clock.
- EVED Mode(For internal test only) (BTDM_CTRL_MODEM_SLEEP_MODE_EVED)
  EVED mode is for BLE only and is only for internal test. Do not use it for production. this mode is not compatible with DFS nor light sleep

CONFIG_BTDM_CTRL_LOW_POWER_CLOCK

Bluetooth low power clock

Found in: Component config > Bluetooth > Controller Options > MODEM SLEEP Options

Select the low power clock source for bluetooth controller. Bluetooth low power clock is the clock source to maintain time in sleep mode.

- "Main crystal" option provides good accuracy and can support Dynamic Frequency Scaling to be used with Bluetooth modem sleep. Light sleep is not supported.
- "External 32kHz crystal" option allows user to use a 32.768kHz crystal as Bluetooth low power clock. This option is allowed as long as External 32kHz crystal is configured as the system RTC clock source. This option provides good accuracy and supports Bluetooth modem sleep to be used alongside Dynamic Frequency Scaling or light sleep.

Available options:

- Main crystal (BTDM_CTRL_LPCLK_SEL_MAIN_XTAL)
  Main crystal can be used as low power clock for bluetooth modem sleep. If this option is selected, bluetooth modem sleep can work under Dynamic Frequency Scaling(DFS) enabled, but cannot work when light sleep is enabled. Main crystal has a good performance in accuracy as the bluetooth low power clock source.
- External 32kHz crystal (BTDM_CTRL_LPCLK_SEL_EXT_32K_XTAL)
  External 32kHz crystal has a nominal frequency of 32.768kHz and provides good frequency stability. If used as Bluetooth low power clock, External 32kHz can support Bluetooth modem sleep to be used with both DFS and light sleep.

CONFIG_BTDM_BLE_SLEEP_CLOCK_ACCURACY

BLE Sleep Clock Accuracy

Found in: Component config > Bluetooth > Controller Options

BLE Sleep Clock Accuracy(SCA) for the local device is used to estimate window widening in BLE connection events. With a lower level of clock accuracy(e.g. 500ppm over 250ppm), the slave needs a larger RX window to synchronize with master in each anchor point, thus resulting in an increase of power consumption but a higher level of robustness in keeping connected. According to the requirements of Bluetooth Core specification 4.2, the worst-case accuracy of Classic Bluetooth low power oscialltor(LPO) is +/-250ppm in STANDBY and in low power modes such as sniff. For BLE the worst-case SCA is +/-500ppm.

- "151ppm to 250ppm" option is the default value for Bluetooth Dual mode
- **"251ppm to 500ppm" option can be used in BLE only mode when using external 32kHz crystal as** low power clock. This option is provided in case that BLE sleep clock has a lower level of accuracy, or other error sources contribute to the inaccurate timing during sleep.

Available options:

- 251ppm to 500ppm (BTDM_BLE_DEFAULT_SCA_500PPM)
- 151ppm to 250ppm (BTDM_BLE_DEFAULT_SCA_250PPM)

CONFIG_BTDM_BLE_SCAN_DUPL

BLE Scan Duplicate Options

Found in: Component config > Bluetooth > Controller Options

This select enables parameters setting of BLE scan duplicate.

Default value:

- Yes (enabled) if (BTDM_CTRL_MODE_BTDM || BTDM_CTRL_MODE_BLE_ONLY) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_SCAN_DUPL_TYPE

Scan Duplicate Type

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_SCAN_DUPL

Scan duplicate have three ways. one is "Scan Duplicate By Device Address", This way is to use advertiser address filtering. The adv packet of the same address is only allowed to be reported once. Another way is "Scan Duplicate By Device Address And Advertising Data" . This way is to use advertising data and device address filtering. All different adv packets with the same address are allowed to be reported. The last way is "Scan Duplicate By Advertising Data" . This way is to use advertising data filtering. All same advertising data only allow to be reported once even though they are from different devices.

Available options:

- Scan Duplicate By Device Address (BTDM_SCAN_DUPL_TYPE_DEVICE)
  This way is to use advertiser address filtering. The adv packet of the same address is only allowed to be reported once
- Scan Duplicate By Advertising Data (BTDM_SCAN_DUPL_TYPE_DATA)
  This way is to use advertising data filtering. All same advertising data only allow to be reported once even though they are from different devices.
- Scan Duplicate By Device Address And Advertising Data (BTDM_SCAN_DUPL_TYPE_DATA_DEVICE)
  This way is to use advertising data and device address filtering. All different adv packets with the same address are allowed to be reported.

CONFIG_BTDM_SCAN_DUPL_CACHE_SIZE

Maximum number of devices in scan duplicate filter

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_SCAN_DUPL

Maximum number of devices which can be recorded in scan duplicate filter. When the maximum amount of device in the filter is reached, the oldest device will be refreshed.

Range:

- from 10 to 1000 if *CONFIG_BTDM_BLE_SCAN_DUPL* && BT_CONTROLLER_ENABLED

Default value:

- 100 if *CONFIG_BTDM_BLE_SCAN_DUPL* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_SCAN_DUPL_CACHE_REFRESH_PERIOD

Duplicate scan list refresh period (seconds)

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_SCAN_DUPL

If the period value is non-zero, the controller will periodically clear the device information stored in the scan duuplicate filter. If it is 0, the scan duuplicate filter will not be cleared until the scanning is disabled. Duplicate advertisements for this period should not be sent to the Host in advertising report events. There are two scenarios where the ADV packet will be repeatedly reported: 1. The duplicate scan cache is full, the controller will delete the oldest device information and add new device information. 2. When the refresh period is up, the controller will clear all device information and start filtering again.

Range:

- from 0 to 1000 if *CONFIG_BTDM_BLE_SCAN_DUPL* && BT_CONTROLLER_ENABLED

Default value:

- 0 if *CONFIG_BTDM_BLE_SCAN_DUPL* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN

Special duplicate scan mechanism for BLE Mesh scan

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_SCAN_DUPL

This enables the BLE scan duplicate for special BLE Mesh scan.

Default value:

- No (disabled) if *CONFIG_BTDM_BLE_SCAN_DUPL* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_MESH_DUPL_SCAN_CACHE_SIZE

Maximum number of Mesh adv packets in scan duplicate filter

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_SCAN_DUPL > CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN

Maximum number of adv packets which can be recorded in duplicate scan cache for BLE Mesh. When the maximum amount of device in the filter is reached, the cache will be refreshed.

Range:

- from 10 to 1000 if *CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN* && BT_CONTROLLER_ENABLED

Default value:

- 100 if *CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED

BLE full scan feature supported

Found in: Component config > Bluetooth > Controller Options

The full scan function is mainly used to provide BLE scan performance. This is required for scenes with high scan performance requirements, such as BLE Mesh scenes.

Default value:

- Yes (enabled) if (BTDM_CTRL_MODE_BLE_ONLY || BTDM_CTRL_MODE_BTDM) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_SCAN_BACKOFF_UPPERLIMITMAX

Disable active scan backoff

Found in: Component config > Bluetooth > Controller Options

Disable active scan backoff. The bluetooth spec requires that scanners should run a backoff procedure to minimize collision of scan request PDUs from nultiple scanners. If scan backoff is disabled, in active scanning, scan request PDU will be sent every time when HW receives scannable ADV PDU.

Default value:

- No (disabled) if BT_CONTROLLER_ENABLED

CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP

BLE adv report flow control supported

Found in: Component config > Bluetooth > Controller Options

The function is mainly used to enable flow control for advertising reports. When it is enabled, advertising reports will be discarded by the controller if the number of unprocessed advertising reports exceeds the size of BLE adv report flow control.

Default value:

- Yes (enabled) if (BTDM_CTRL_MODE_BTDM || BTDM_CTRL_MODE_BLE_ONLY) && BT_CONTROLLER_ENABLED

CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM

BLE adv report flow control number

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP

The number of unprocessed advertising report that Bluedroid can save.If you set *BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM* to a small value, this may cause adv packets lost. If you set *BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM* to a large value, Bluedroid may cache a lot of adv packets and this may cause system memory run out. For example, if you set it to 50, the maximum memory consumed by host is 35 * 50 bytes. Please set *BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM* according to your system free memory and handle adv packets as fast as possible, otherwise it will cause adv packets lost.

Range:

- from 50 to 1000 if *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP* && BT_CONTROLLER_ENABLED

Default value:

- 100 if *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD

BLE adv lost event threshold value

Found in: Component config > Bluetooth > Controller Options > CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP

When adv report flow control is enabled, The ADV lost event will be generated when the number of ADV packets lost in the controller reaches this threshold. It is better to set a larger value. If you set *BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD* to a small value or printf every adv lost event, it may cause adv packets lost more.

Range:

- from 1 to 1000 if *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP* && BT_CONTROLLER_ENABLED

Default value:

- 20 if *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP* && BT_CONTROLLER_ENABLED

CONFIG_BTDM_CTRL_HLI

High level interrupt

Found in: Component config > Bluetooth > Controller Options

Using Level 4 interrupt for Bluetooth.

Default value:

- Yes (enabled) if *CONFIG_BT_ENABLED* && BT_CONTROLLER_ENABLED

CONFIG_BT_RELEASE_IRAM

Release Bluetooth text (READ DOCS FIRST)

Found in: Component config > Bluetooth

This option release Bluetooth text section and merge Bluetooth data, bss & text into a large free heap region when esp_bt_mem_release is called, total saving ~21kB or more of IRAM. ESP32-C2 only 3 configurable PMP entries available, rest of them are hard-coded. We cannot split the memory into 3 different regions (IRAM, BLE-IRAM, DRAM). So this option will disable the PMP (ESP_SYSTEM_PMP_IDRAM_SPLIT)

Default value:

- No (disabled) if *CONFIG_BT_ENABLED* && BT_LE_RELEASE_IRAM_SUPPORTED

CONFIG_BT_HCI_LOG_DEBUG_EN

Enable Bluetooth HCI debug mode

Found in: Component config > Bluetooth

This option is used to enable bluetooth debug mode, which saves the hci layer data stream.

Default value:

- No (disabled) if BT_BLUEDROID_ENABLED || BT_NIMBLE_ENABLED

CONFIG_BT_HCI_LOG_DATA_BUFFER_SIZE

Size of the cache used for HCI data in Bluetooth HCI debug mode (N*1024 bytes)

Found in: Component config > Bluetooth > CONFIG_BT_HCI_LOG_DEBUG_EN

This option is to configure the buffer size of the hci data steam cache in hci debug mode. This is a ring buffer, the new data will overwrite the oldest data if the buffer is full.

Range:

- from 1 to 100 if *CONFIG_BT_HCI_LOG_DEBUG_EN*

Default value:

- 5 if *CONFIG_BT_HCI_LOG_DEBUG_EN*

CONFIG_BT_HCI_LOG_ADV_BUFFER_SIZE

Size of the cache used for adv report in Bluetooth HCI debug mode (N*1024 bytes)

Found in: Component config > Bluetooth > CONFIG_BT_HCI_LOG_DEBUG_EN

This option is to configure the buffer size of the hci adv report cache in hci debug mode. This is a ring buffer, the new data will overwrite the oldest data if the buffer is full.

Range:

- from 1 to 100 if *CONFIG_BT_HCI_LOG_DEBUG_EN*

Default value:

- 8 if *CONFIG_BT_HCI_LOG_DEBUG_EN*

CONFIG_BLE_MESH

ESP BLE Mesh Support

Found in: Component config

This option enables ESP BLE Mesh support. The specific features that are available may depend on other features that have been enabled in the stack, such as Bluetooth Support, Bluedroid Support & GATT support.

Contains:

- *BLE Mesh and BLE coexistence support*
- *CONFIG_BLE_MESH_GATT_PROXY_CLIENT*
- *CONFIG_BLE_MESH_GATT_PROXY_SERVER*
- *BLE Mesh NET BUF DEBUG LOG LEVEL*
- *CONFIG_BLE_MESH_PROV*
- *CONFIG_BLE_MESH_PROXY*
- *BLE Mesh specific test option*
- *BLE Mesh STACK DEBUG LOG LEVEL*
- *CONFIG_BLE_MESH_NO_LOG*
- *CONFIG_BLE_MESH_IVU_DIVIDER*

- *CONFIG_BLE_MESH_FAST_PROV*
- *CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC*
- *CONFIG_BLE_MESH_EXPERIMENTAL*
- *CONFIG_BLE_MESH_CRPL*
- *CONFIG_BLE_MESH_RX_SDU_MAX*
- *CONFIG_BLE_MESH_MODEL_KEY_COUNT*
- *CONFIG_BLE_MESH_APP_KEY_COUNT*
- *CONFIG_BLE_MESH_MODEL_GROUP_COUNT*
- *CONFIG_BLE_MESH_LABEL_COUNT*
- *CONFIG_BLE_MESH_SUBNET_COUNT*
- *CONFIG_BLE_MESH_TX_SEG_MAX*
- *CONFIG_BLE_MESH_RX_SEG_MSG_COUNT*
- *CONFIG_BLE_MESH_TX_SEG_MSG_COUNT*
- *CONFIG_BLE_MESH_MEM_ALLOC_MODE*
- *CONFIG_BLE_MESH_MSG_CACHE_SIZE*
- *CONFIG_BLE_MESH_NOT_RELAY_REPLAY_MSG*
- *CONFIG_BLE_MESH_ADV_BUF_COUNT*
- *CONFIG_BLE_MESH_PB_GATT*
- *CONFIG_BLE_MESH_PB_ADV*
- *CONFIG_BLE_MESH_IVU_RECOVERY_IVI*
- *CONFIG_BLE_MESH_RELAY*
- *CONFIG_BLE_MESH_SETTINGS*
- *CONFIG_BLE_MESH_DEINIT*
- *CONFIG_BLE_MESH_USE_DUPLICATE_SCAN*
- *Support for BLE Mesh Client/Server models*
- *Support for BLE Mesh Foundation models*
- *CONFIG_BLE_MESH_NODE*
- *CONFIG_BLE_MESH_PROVISIONER*
- *CONFIG_BLE_MESH_FRIEND*
- *CONFIG_BLE_MESH_LOW_POWER*
- *CONFIG_BLE_MESH_HCI_5_0*
- *CONFIG_BLE_MESH_RANDOM_ADV_INTERVAL*
- *CONFIG_BLE_MESH_IV_UPDATE_TEST*
- *CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT*

CONFIG_BLE_MESH_HCI_5_0

Support sending 20ms non-connectable adv packets

Found in: Component config > CONFIG_BLE_MESH

It is a temporary solution and needs further modifications.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RANDOM_ADV_INTERVAL

Support using random adv interval for mesh packets

Found in: Component config > CONFIG_BLE_MESH

Enable this option to allow using random advertising interval for mesh packets. And this could help avoid collision of advertising packets.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_USE_DUPLICATE_SCAN

Support Duplicate Scan in BLE Mesh

Found in: Component config > CONFIG_BLE_MESH

Enable this option to allow using specific duplicate scan filter in BLE Mesh, and Scan Duplicate Type must be set by choosing the option in the Bluetooth Controller section in menuconfig, which is "Scan Duplicate By Device Address and Advertising Data".

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MEM_ALLOC_MODE

Memory allocation strategy

Found in: Component config > CONFIG_BLE_MESH

Allocation strategy for BLE Mesh stack, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only
- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF
- Internal IRAM memory wherever applicable else internal DRAM

Recommended mode here is always internal (*), since that is most preferred from security perspective. But if application requirement does not allow sufficient free internal memory then alternate mode can be selected.

(*) In case of ESP32-S2/ESP32-S3, hardware allows encryption of external SPIRAM contents provided hardware flash encryption feature is enabled. In that case, using external SPIRAM allocation strategy is also safe choice from security perspective.

Available options:

- Internal DRAM (BLE_MESH_MEM_ALLOC_MODE_INTERNAL)
- External SPIRAM (BLE_MESH_MEM_ALLOC_MODE_EXTERNAL)
- Default alloc mode (BLE_MESH_MEM_ALLOC_MODE_DEFAULT)
  Enable this option to use the default memory allocation strategy when external SPIRAM is enabled. See the SPIRAM options for more details.
- Internal IRAM (BLE_MESH_MEM_ALLOC_MODE_IRAM_8BIT)
  Allows to use IRAM memory region as 8bit accessible region. Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC

Enable FreeRTOS static allocation

Found in: Component config > CONFIG_BLE_MESH

Enable this option to use FreeRTOS static allocation APIs for BLE Mesh, which provides the ability to use different dynamic memory (i.e. SPIRAM or IRAM) for FreeRTOS objects. If this option is disabled, the FreeRTOS static allocation APIs will not be used, and internal DRAM will be allocated for FreeRTOS objects.

Default value:

- No (disabled) if (*CONFIG_SPIRAM* || *CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY*) && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC_MODE

Memory allocation for FreeRTOS objects

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC

Choose the memory to be used for FreeRTOS objects.

Available options:

- External SPIRAM (BLE_MESH_FREERTOS_STATIC_ALLOC_EXTERNAL)
  If enabled, BLE Mesh allocates dynamic memory from external SPIRAM for FreeRTOS objects, i.e. mutex, queue, and task stack. External SPIRAM can only be used for task stack when SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY is enabled. See the SPIRAM options for more details.
- Internal IRAM (BLE_MESH_FREERTOS_STATIC_ALLOC_IRAM_8BIT)
  If enabled, BLE Mesh allocates dynamic memory from internal IRAM for FreeRTOS objects, i.e. mutex, queue. Note: IRAM region cannot be used as task stack.

CONFIG_BLE_MESH_DEINIT

Support de-initialize BLE Mesh stack

Found in: Component config > CONFIG_BLE_MESH

If enabled, users can use the function esp_ble_mesh_deinit() to de-initialize the whole BLE Mesh stack.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

BLE Mesh and BLE coexistence support

Contains:

- *CONFIG_BLE_MESH_SUPPORT_BLE_SCAN*
- *CONFIG_BLE_MESH_SUPPORT_BLE_ADV*

CONFIG_BLE_MESH_SUPPORT_BLE_ADV

Support sending normal BLE advertising packets

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh and BLE coexistence support

When selected, users can send normal BLE advertising packets with specific API.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_BLE_ADV_BUF_COUNT

Number of advertising buffers for BLE advertising packets

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh and BLE coexistence support > CONFIG_BLE_MESH_SUPPORT_BLE_ADV

Number of advertising buffers for BLE packets available.

Range:

- from 1 to 255 if *CONFIG_BLE_MESH_SUPPORT_BLE_ADV* && *CONFIG_BLE_MESH*

Default value:

- 3 if *CONFIG_BLE_MESH_SUPPORT_BLE_ADV* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SUPPORT_BLE_SCAN

Support scanning normal BLE advertising packets

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh and BLE coexistence support

When selected, users can register a callback and receive normal BLE advertising packets in the application layer.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FAST_PROV

Enable BLE Mesh Fast Provisioning

Found in: Component config > CONFIG_BLE_MESH

Enable this option to allow BLE Mesh fast provisioning solution to be used. When there are multiple unprovisioned devices around, fast provisioning can greatly reduce the time consumption of the whole provisioning process. When this option is enabled, and after an unprovisioned device is provisioned into a node successfully, it can be changed to a temporary Provisioner.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_NODE

Support for BLE Mesh Node

Found in: Component config > CONFIG_BLE_MESH

Enable the device to be provisioned into a node. This option should be enabled when an unprovisioned device is going to be provisioned into a node and communicate with other nodes in the BLE Mesh network.

CONFIG_BLE_MESH_PROVISIONER

Support for BLE Mesh Provisioner

Found in: Component config > CONFIG_BLE_MESH

Enable the device to be a Provisioner. The option should be enabled when a device is going to act as a Provisioner and provision unprovisioned devices into the BLE Mesh network.

CONFIG_BLE_MESH_WAIT_FOR_PROV_MAX_DEV_NUM

Maximum number of unprovisioned devices that can be added to device queue

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROVISIONER

This option specifies how many unprovisioned devices can be added to device queue for provisioning. Users can use this option to define the size of the queue in the bottom layer which is used to store unprovisioned device information (e.g. Device UUID, address).

Range:

- from 1 to 100 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 10 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MAX_PROV_NODES

Maximum number of devices that can be provisioned by Provisioner

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROVISIONER

This option specifies how many devices can be provisioned by a Provisioner. This value indicates the maximum number of unprovisioned devices which can be provisioned by a Provisioner. For instance, if the value is 6, it means the Provisioner can provision up to 6 unprovisioned devices. Theoretically a Provisioner without the limitation of its memory can provision up to 32766 unprovisioned devices, here we limit the maximum number to 100 just to limit the memory used by a Provisioner. The bigger the value is, the more memory it will cost by a Provisioner to store the information of nodes.

Range:

- from 1 to 1000 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 10 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PBA_SAME_TIME

Maximum number of PB-ADV running at the same time by Provisioner

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_PROVISIONER*

This option specifies how many devices can be provisioned at the same time using PB-ADV. For examples, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-ADV at the same time.

Range:

- from 1 to 10 if *CONFIG_BLE_MESH_PB_ADV* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 2 if *CONFIG_BLE_MESH_PB_ADV* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PBG_SAME_TIME

Maximum number of PB-GATT running at the same time by Provisioner

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_PROVISIONER*

This option specifies how many devices can be provisioned at the same time using PB-GATT. For example, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-GATT at the same time.

Range:

- from 1 to 5 if *CONFIG_BLE_MESH_PB_GATT* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 1 if *CONFIG_BLE_MESH_PB_GATT* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROVISIONER_SUBNET_COUNT

Maximum number of mesh subnets that can be created by Provisioner

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_PROVISIONER*

This option specifies how many subnets per network a Provisioner can create. Indeed, this value decides the number of network keys which can be added by a Provisioner.

Range:

- from 1 to 4096 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 3 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROVISIONER_APP_KEY_COUNT

Maximum number of application keys that can be owned by Provisioner

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_PROVISIONER*

This option specifies how many application keys the Provisioner can have. Indeed, this value decides the number of the application keys which can be added by a Provisioner.

Range:

- from 1 to 4096 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 3 if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROVISIONER_RECV_HB

Support receiving Heartbeat messages

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROVISIONER

When this option is enabled, Provisioner can call specific functions to enable or disable receiving Heartbeat messages and notify them to the application layer.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROVISIONER_RECV_HB_FILTER_SIZE

Maximum number of filter entries for receiving Heartbeat messages

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROVISIONER > CONFIG_BLE_MESH_PROVISIONER_RECV_HB

This option specifies how many heartbeat filter entries Provisioner supports. The heartbeat filter (acceptlist or rejectlist) entries are used to store a list of SRC and DST which can be used to decide if a heartbeat message will be processed and notified to the application layer by Provisioner. Note: The filter is an empty rejectlist by default.

Range:

- from 1 to 1000 if *CONFIG_BLE_MESH_PROVISIONER_RECV_HB* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

Default value:

- 3 if *CONFIG_BLE_MESH_PROVISIONER_RECV_HB* && *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROV

BLE Mesh Provisioning support

Found in: Component config > CONFIG_BLE_MESH

Enable this option to support BLE Mesh Provisioning functionality. For BLE Mesh, this option should be always enabled.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PB_ADV

Provisioning support using the advertising bearer (PB-ADV)

Found in: Component config > CONFIG_BLE_MESH

Enable this option to allow the device to be provisioned over the advertising bearer. This option should be enabled if PB-ADV is going to be used during provisioning procedure.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_UNPROVISIONED_BEACON_INTERVAL

Interval between two consecutive Unprovisioned Device Beacon

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PB_ADV

This option specifies the interval of sending two consecutive unprovisioned device beacon, users can use this option to change the frequency of sending unprovisioned device beacon. For example, if the value is 5, it means the unprovisioned device beacon will send every 5 seconds. When the option of BLE_MESH_FAST_PROV is selected, the value is better to be 3 seconds, or less.

Range:

- from 1 to 100 if *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH_PB_ADV* && *CONFIG_BLE_MESH*

Default value:

- 5 if *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH_PB_ADV* && *CONFIG_BLE_MESH*
- 3 if *CONFIG_BLE_MESH_FAST_PROV* && *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH_PB_ADV* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PB_GATT

Provisioning support using GATT (PB-GATT)

Found in: Component config > CONFIG_BLE_MESH

Enable this option to allow the device to be provisioned over GATT. This option should be enabled if PB-GATT is going to be used during provisioning procedure.

Virtual option enabled whenever any Proxy protocol is needed

CONFIG_BLE_MESH_PROXY

BLE Mesh Proxy protocol support

Found in: Component config > CONFIG_BLE_MESH

Enable this option to support BLE Mesh Proxy protocol used by PB-GATT and other proxy pdu transmission.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_GATT_PROXY_SERVER

BLE Mesh GATT Proxy Server

Found in: Component config > CONFIG_BLE_MESH

This option enables support for Mesh GATT Proxy Service, i.e. the ability to act as a proxy between a Mesh GATT Client and a Mesh network. This option should be enabled if a node is going to be a Proxy Server.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_NODE_ID_TIMEOUT

Node Identity advertising timeout

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_SERVER

This option determines for how long the local node advertises using Node Identity. The given value is in seconds. The specification limits this to 60 seconds and lists it as the recommended value as well. So leaving the default value is the safest option. When an unprovisioned device is provisioned successfully

and becomes a node, it will start to advertise using Node Identity during the time set by this option. And after that, Network ID will be advertised.

Range:
- from 1 to 60 if *CONFIG_BLE_MESH_GATT_PROXY_SERVER* && *CONFIG_BLE_MESH*

Default value:
- 60 if *CONFIG_BLE_MESH_GATT_PROXY_SERVER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PROXY_FILTER_SIZE

Maximum number of filter entries per Proxy Client

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_SERVER

This option specifies how many Proxy Filter entries the local node supports. The entries of Proxy filter (whitelist or blacklist) are used to store a list of addresses which can be used to decide which messages will be forwarded to the Proxy Client by the Proxy Server.

Range:
- from 1 to 32767 if *CONFIG_BLE_MESH_GATT_PROXY_SERVER* && *CONFIG_BLE_MESH*

Default value:
- 4 if *CONFIG_BLE_MESH_GATT_PROXY_SERVER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_GATT_PROXY_CLIENT

BLE Mesh GATT Proxy Client

Found in: Component config > CONFIG_BLE_MESH

This option enables support for Mesh GATT Proxy Client. The Proxy Client can use the GATT bearer to send mesh messages to a node that supports the advertising bearer.

Default value:
- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SETTINGS

Store BLE Mesh configuration persistently

Found in: Component config > CONFIG_BLE_MESH

When selected, the BLE Mesh stack will take care of storing/restoring the BLE Mesh configuration persistently in flash. If the device is a BLE Mesh node, when this option is enabled, the configuration of the device will be stored persistently, including unicast address, NetKey, AppKey, etc. And if the device is a BLE Mesh Provisioner, the information of the device will be stored persistently, including the information of provisioned nodes, NetKey, AppKey, etc.

Default value:
- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_STORE_TIMEOUT

Delay (in seconds) before storing anything persistently

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_SETTINGS

This value defines in seconds how soon any pending changes are actually written into persistent storage (flash) after a change occurs. The option allows nodes to delay a certain period of time to save proper information to flash. The default value is 0, which means information will be stored immediately once there are updates.

Range:
- from 0 to 1000000 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

Default value:

- 0 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SEQ_STORE_RATE

How often the sequence number gets updated in storage

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_SETTINGS*

This value defines how often the local sequence number gets updated in persistent storage (i.e. flash). e.g. a value of 100 means that the sequence number will be stored to flash on every 100th increment. If the node sends messages very frequently a higher value makes more sense, whereas if the node sends infrequently a value as low as 0 (update storage for every increment) can make sense. When the stack gets initialized it will add sequence number to the last stored one, so that it starts off with a value that's guaranteed to be larger than the last one used before power off.

Range:

- from 0 to 1000000 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

Default value:

- 0 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RPL_STORE_TIMEOUT

Minimum frequency that the RPL gets updated in storage

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_SETTINGS*

This value defines in seconds how soon the RPL (Replay Protection List) gets written to persistent storage after a change occurs. If the node receives messages frequently, then a large value is recommended. If the node receives messages rarely, then the value can be as low as 0 (which means the RPL is written into the storage immediately). Note that if the node operates in a security-sensitive case, and there is a risk of sudden power-off, then a value of 0 is strongly recommended. Otherwise, a power loss before RPL being written into the storage may introduce message replay attacks and system security will be in a vulnerable state.

Range:

- from 0 to 1000000 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

Default value:

- 0 if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SETTINGS_BACKWARD_COMPATIBILITY

A specific option for settings backward compatibility

Found in: Component config > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_SETTINGS*

This option is created to solve the issue of failure in recovering node information after mesh stack updates. In the old version mesh stack, there is no key of "mesh/role" in nvs. In the new version mesh stack, key of "mesh/role" is added in nvs, recovering node information needs to check "mesh/role" key in nvs and implements selective recovery of mesh node information. Therefore, there may be failure in recovering node information during node restarting after OTA.

The new version mesh stack adds the option of "mesh/role" because we have added the support of storing Provisioner information, while the old version only supports storing node information.

If users are updating their nodes from old version to new version, we recommend enabling this option, so that system could set the flag in advance before recovering node information and make sure the node information recovering could work as expected.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SPECIFIC_PARTITION

Use a specific NVS partition for BLE Mesh

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_SETTINGS

When selected, the mesh stack will use a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using nvs_flash_init_partition() API, and the partition must exists in the csv file. When Provisioner needs to store a large amount of nodes' information in the flash (e.g. more than 20), this option is recommended to be enabled.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_PARTITION_NAME

Name of the NVS partition for BLE Mesh

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_SETTINGS > CONFIG_BLE_MESH_SPECIFIC_PARTITION

This value defines the name of the specified NVS partition used by the mesh stack.

Default value:

- "ble_mesh" if *CONFIG_BLE_MESH_SPECIFIC_PARTITION* && *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE

Support using multiple NVS namespaces by Provisioner

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_SETTINGS

When selected, Provisioner can use different NVS namespaces to store different instances of mesh information. For example, if in the first room, Provisioner uses NetKey A, AppKey A and provisions three devices, these information will be treated as mesh information instance A. When the Provisioner moves to the second room, it uses NetKey B, AppKey B and provisions two devices, then the information will be treated as mesh information instance B. Here instance A and instance B will be stored in different namespaces. With this option enabled, Provisioner needs to use specific functions to open the corresponding NVS namespace, restore the mesh information, release the mesh information or erase the mesh information.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_PROVISIONER* && *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MAX_NVS_NAMESPACE

Maximum number of NVS namespaces

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_SETTINGS > CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE

This option specifies the maximum NVS namespaces supported by Provisioner.

Range:

- from 1 to 255 if *CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE* && *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

Default value:

- 2 if *CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE* && *CONFIG_BLE_MESH_SETTINGS* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SUBNET_COUNT

Maximum number of mesh subnets per network

Found in: Component config > CONFIG_BLE_MESH

This option specifies how many subnets a Mesh network can have at the same time. Indeed, this value decides the number of the network keys which can be owned by a node.

Range:
- from 1 to 4096 if *CONFIG_BLE_MESH*

Default value:
- 3 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_APP_KEY_COUNT

Maximum number of application keys per network

Found in: Component config > CONFIG_BLE_MESH

This option specifies how many application keys the device can store per network. Indeed, this value decides the number of the application keys which can be owned by a node.

Range:
- from 1 to 4096 if *CONFIG_BLE_MESH*

Default value:
- 3 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MODEL_KEY_COUNT

Maximum number of application keys per model

Found in: Component config > CONFIG_BLE_MESH

This option specifies the maximum number of application keys to which each model can be bound.

Range:
- from 1 to 4096 if *CONFIG_BLE_MESH*

Default value:
- 3 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MODEL_GROUP_COUNT

Maximum number of group address subscriptions per model

Found in: Component config > CONFIG_BLE_MESH

This option specifies the maximum number of addresses to which each model can be subscribed.

Range:
- from 1 to 4096 if *CONFIG_BLE_MESH*

Default value:
- 3 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LABEL_COUNT

Maximum number of Label UUIDs used for Virtual Addresses

Found in: Component config > CONFIG_BLE_MESH

This option specifies how many Label UUIDs can be stored. Indeed, this value decides the number of the Virtual Addresses can be supported by a node.

Range:
- from 0 to 4096 if *CONFIG_BLE_MESH*

Default value:

- 3 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_CRPL

Maximum capacity of the replay protection list

Found in: Component config > CONFIG_BLE_MESH

This option specifies the maximum capacity of the replay protection list. It is similar to Network message cache size, but has a different purpose. The replay protection list is used to prevent a node from replay attack, which will store the source address and sequence number of the received mesh messages. For Provisioner, the replay protection list size should not be smaller than the maximum number of nodes whose information can be stored. And the element number of each node should also be taken into consideration. For example, if Provisioner can provision up to 20 nodes and each node contains two elements, then the replay protection list size of Provisioner should be at least 40.

Range:

- from 2 to 65535 if *CONFIG_BLE_MESH*

Default value:

- 10 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_NOT_RELAY_REPLAY_MSG

Not relay replayed messages in a mesh network

Found in: Component config > CONFIG_BLE_MESH

There may be many expired messages in a complex mesh network that would be considered replayed messages. Enable this option will refuse to relay such messages, which could help to reduce invalid packets in the mesh network. However, it should be noted that enabling this option may result in packet loss in certain environments. Therefore, users need to decide whether to enable this option according to the actual usage situation.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_EXPERIMENTAL* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_MSG_CACHE_SIZE

Network message cache size

Found in: Component config > CONFIG_BLE_MESH

Number of messages that are cached for the network. This helps prevent unnecessary decryption operations and unnecessary relays. This option is similar to Replay protection list, but has a different purpose. A node is not required to cache the entire Network PDU and may cache only part of it for tracking, such as values for SRC/SEQ or others.

Range:

- from 2 to 65535 if *CONFIG_BLE_MESH*

Default value:

- 10 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_ADV_BUF_COUNT

Number of advertising buffers

Found in: Component config > CONFIG_BLE_MESH

Number of advertising buffers available. The transport layer reserves ADV_BUF_COUNT - 3 buffers for outgoing segments. The maximum outgoing SDU size is 12 times this value (out of which 4 or 8 bytes are used for the Transport Layer MIC). For example, 5 segments means the maximum SDU size

is 60 bytes, which leaves 56 bytes for application layer data using a 4-byte MIC, or 52 bytes using an 8-byte MIC.

Range:
- from 6 to 256 if *CONFIG_BLE_MESH*

Default value:
- 60 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_IVU_DIVIDER

Divider for IV Update state refresh timer

Found in: Component config > CONFIG_BLE_MESH

When the IV Update state enters Normal operation or IV Update in Progress, we need to keep track of how many hours has passed in the state, since the specification requires us to remain in the state at least for 96 hours (Update in Progress has an additional upper limit of 144 hours).

In order to fulfill the above requirement, even if the node might be powered off once in a while, we need to store persistently how many hours the node has been in the state. This doesn't necessarily need to happen every hour (thanks to the flexible duration range). The exact cadence will depend a lot on the ways that the node will be used and what kind of power source it has.

Since there is no single optimal answer, this configuration option allows specifying a divider, i.e. how many intervals the 96 hour minimum gets split into. After each interval the duration that the node has been in the current state gets stored to flash. E.g. the default value of 4 means that the state is saved every 24 hours (96 / 4).

Range:
- from 2 to 96 if *CONFIG_BLE_MESH*

Default value:
- 4 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_IVU_RECOVERY_IVI

Recovery the IV index when the latest whole IV update procedure is missed

Found in: Component config > CONFIG_BLE_MESH

According to Section 3.10.5 of Mesh Specification v1.0.1. If a node in Normal Operation receives a Secure Network beacon with an IV index equal to the last known IV index+1 and the IV Update Flag set to 0, the node may update its IV without going to the IV Update in Progress state, or it may initiate an IV Index Recovery procedure (Section 3.10.6), or it may ignore the Secure Network beacon. The node makes the choice depending on the time since last IV update and the likelihood that the node has missed the Secure Network beacons with the IV update Flag. When the above situation is encountered, this option can be used to decide whether to perform the IV index recovery procedure.

Default value:
- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_TX_SEG_MSG_COUNT

Maximum number of simultaneous outgoing segmented messages

Found in: Component config > CONFIG_BLE_MESH

Maximum number of simultaneous outgoing multi-segment and/or reliable messages. The default value is 1, which means the device can only send one segmented message at a time. And if another segmented message is going to be sent, it should wait for the completion of the previous one. If users are going to send multiple segmented messages at the same time, this value should be configured properly.

Range:
- from 1 to if *CONFIG_BLE_MESH*

Default value:

- 1 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RX_SEG_MSG_COUNT

Maximum number of simultaneous incoming segmented messages

Found in: Component config > CONFIG_BLE_MESH

Maximum number of simultaneous incoming multi-segment and/or reliable messages. The default value is 1, which means the device can only receive one segmented message at a time. And if another segmented message is going to be received, it should wait for the completion of the previous one. If users are going to receive multiple segmented messages at the same time, this value should be configured properly.

Range:

- from 1 to 255 if *CONFIG_BLE_MESH*

Default value:

- 1 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RX_SDU_MAX

Maximum incoming Upper Transport Access PDU length

Found in: Component config > CONFIG_BLE_MESH

Maximum incoming Upper Transport Access PDU length. Leave this to the default value, unless you really need to optimize memory usage.

Range:

- from 36 to 384 if *CONFIG_BLE_MESH*

Default value:

- 384 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_TX_SEG_MAX

Maximum number of segments in outgoing messages

Found in: Component config > CONFIG_BLE_MESH

Maximum number of segments supported for outgoing messages. This value should typically be fine-tuned based on what models the local node supports, i.e. what's the largest message payload that the node needs to be able to send. This value affects memory and call stack consumption, which is why the default is lower than the maximum that the specification would allow (32 segments).

The maximum outgoing SDU size is 12 times this number (out of which 4 or 8 bytes is used for the Transport Layer MIC). For example, 5 segments means the maximum SDU size is 60 bytes, which leaves 56 bytes for application layer data using a 4-byte MIC and 52 bytes using an 8-byte MIC.

Be sure to specify a sufficient number of advertising buffers when setting this option to a higher value. There must be at least three more advertising buffers (BLE_MESH_ADV_BUF_COUNT) as there are outgoing segments.

Range:

- from 2 to 32 if *CONFIG_BLE_MESH*

Default value:

- 32 if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RELAY

Relay support

Found in: Component config > CONFIG_BLE_MESH

Support for acting as a Mesh Relay Node. Enabling this option will allow a node to support the Relay feature, and the Relay feature can still be enabled or disabled by proper configuration messages. Disabling this option will let a node not support the Relay feature.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH_NODE* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RELAY_ADV_BUF

Use separate advertising buffers for relay packets

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_RELAY

When selected, self-send packets will be put in a high-priority queue and relay packets will be put in a low-priority queue.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_RELAY* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_RELAY_ADV_BUF_COUNT

Number of advertising buffers for relay packets

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_RELAY > CONFIG_BLE_MESH_RELAY_ADV_BUF

Number of advertising buffers for relay packets available.

Range:

- from 6 to 256 if *CONFIG_BLE_MESH_RELAY_ADV_BUF* && *CONFIG_BLE_MESH_RELAY* && *CONFIG_BLE_MESH*

Default value:

- 60 if *CONFIG_BLE_MESH_RELAY_ADV_BUF* && *CONFIG_BLE_MESH_RELAY* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LOW_POWER

Support for Low Power features

Found in: Component config > CONFIG_BLE_MESH

Enable this option to operate as a Low Power Node. If low power consumption is required by a node, this option should be enabled. And once the node enters the mesh network, it will try to find a Friend node and establish a friendship.

CONFIG_BLE_MESH_LPN_ESTABLISHMENT

Perform Friendship establishment using low power

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Perform the Friendship establishment using low power with the help of a reduced scan duty cycle. The downside of this is that the node may miss out on messages intended for it until it has successfully set up Friendship with a Friend node. When this option is enabled, the node will stop scanning for a period of time after a Friend Request or Friend Poll is sent, so as to reduce more power consumption.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_AUTO

Automatically start looking for Friend nodes once provisioned

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Once provisioned, automatically enable LPN functionality and start looking for Friend nodes. If this option is disabled LPN mode needs to be manually enabled by calling bt_mesh_lpn_set(true). When an unprovisioned device is provisioned successfully and becomes a node, enabling this option will trigger the node starts to send Friend Request at a certain period until it finds a proper Friend node.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_AUTO_TIMEOUT

Time from last received message before going to LPN mode

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER > CONFIG_BLE_MESH_LPN_AUTO

Time in seconds from the last received message, that the node waits out before starting to look for Friend nodes.

Range:

- from 0 to 3600 if *CONFIG_BLE_MESH_LPN_AUTO* && *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:

- 15 if *CONFIG_BLE_MESH_LPN_AUTO* && *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_RETRY_TIMEOUT

Retry timeout for Friend requests

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Time in seconds between Friend Requests, if a previous Friend Request did not yield any acceptable Friend Offers.

Range:

- from 1 to 3600 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:

- 6 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_RSSI_FACTOR

RSSIFactor, used in Friend Offer Delay calculation

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

The contribution of the RSSI, measured by the Friend node, used in Friend Offer Delay calculations. 0 = 1, 1 = 1.5, 2 = 2, 3 = 2.5. RSSIFactor, one of the parameters carried by Friend Request sent by Low Power node, which is used to calculate the Friend Offer Delay.

Range:

- from 0 to 3 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:

- 0 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_RECV_WIN_FACTOR

ReceiveWindowFactor, used in Friend Offer Delay calculation

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

The contribution of the supported Receive Window used in Friend Offer Delay calculations. 0 = 1, 1 = 1.5, 2 = 2, 3 = 2.5. ReceiveWindowFactor, one of the parameters carried by Friend Request sent by Low Power node, which is used to calculate the Friend Offer Delay.

Range:
- from 0 to 3 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 0 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_MIN_QUEUE_SIZE

Minimum size of the acceptable friend queue (MinQueueSizeLog)

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

The MinQueueSizeLog field is defined as log_2(N), where N is the minimum number of maximum size Lower Transport PDUs that the Friend node can store in its Friend Queue. As an example, MinQueueSizeLog value 1 gives N = 2, and value 7 gives N = 128.

Range:
- from 1 to 7 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 1 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_RECV_DELAY

Receive delay requested by the local node

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

The ReceiveDelay is the time between the Low Power node sending a request and listening for a response. This delay allows the Friend node time to prepare the response. The value is in units of milliseconds.

Range:
- from 10 to 255 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 100 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_POLL_TIMEOUT

The value of the PollTimeout timer

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

PollTimeout timer is used to measure time between two consecutive requests sent by a Low Power node. If no requests are received the Friend node before the PollTimeout timer expires, then the friendship is considered terminated. The value is in units of 100 milliseconds, so e.g. a value of 300 means 30 seconds. The smaller the value, the faster the Low Power node tries to get messages from corresponding Friend node and vice versa.

Range:
- from 10 to 244735 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 300 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_INIT_POLL_TIMEOUT

The starting value of the PollTimeout timer

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

The initial value of the PollTimeout timer when Friendship is to be established for the first time. After this, the timeout gradually grows toward the actual PollTimeout, doubling in value for each iteration. The value is in units of 100 milliseconds, so e.g. a value of 300 means 30 seconds.

Range:
- from 10 to if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_SCAN_LATENCY

Latency for enabling scanning

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Latency (in milliseconds) is the time it takes to enable scanning. In practice, it means how much time in advance of the Receive Window, the request to enable scanning is made.

Range:
- from 0 to 50 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 10 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_GROUPS

Number of groups the LPN can subscribe to

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Maximum number of groups to which the LPN can subscribe.

Range:
- from 0 to 16384 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

Default value:
- 8 if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LPN_SUB_ALL_NODES_ADDR

Automatically subscribe all nodes address

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER

Automatically subscribe all nodes address when friendship established.

Default value:
- No (disabled) if *CONFIG_BLE_MESH_LOW_POWER* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FRIEND

Support for Friend feature

Found in: Component config > CONFIG_BLE_MESH

Enable this option to be able to act as a Friend Node.

CONFIG_BLE_MESH_FRIEND_RECV_WIN

Friend Receive Window

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FRIEND

Receive Window in milliseconds supported by the Friend node.

Range:
- from 1 to 255 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

Default value:
- 255 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FRIEND_QUEUE_SIZE

Minimum number of buffers supported per Friend Queue

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FRIEND

Minimum number of buffers available to be stored for each local Friend Queue. This option decides the size of each buffer which can be used by a Friend node to store messages for each Low Power node.

Range:
- from 2 to 65536 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

Default value:
- 16 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FRIEND_SUB_LIST_SIZE

Friend Subscription List Size

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FRIEND

Size of the Subscription List that can be supported by a Friend node for a Low Power node. And Low Power node can send Friend Subscription List Add or Friend Subscription List Remove messages to the Friend node to add or remove subscription addresses.

Range:
- from 0 to 1023 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

Default value:
- 3 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FRIEND_LPN_COUNT

Number of supported LPN nodes

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FRIEND

Number of Low Power Nodes with which a Friend can have Friendship simultaneously. A Friend node can have friendship with multiple Low Power nodes at the same time, while a Low Power node can only establish friendship with only one Friend node at the same time.

Range:
- from 1 to 1000 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

Default value:
- 2 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_FRIEND_SEG_RX

Number of incomplete segment lists per LPN

Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_FRIEND

Number of incomplete segment lists tracked for each Friends' LPN. In other words, this determines from how many elements can segmented messages destined for the Friend queue be received simultaneously.

Range:
- from 1 to 1000 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

Default value:
- 1 if *CONFIG_BLE_MESH_FRIEND* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_NO_LOG

Disable BLE Mesh debug logs (minimize bin size)

Found in: Component config > CONFIG_BLE_MESH

Select this to save the BLE Mesh related rodata code size. Enabling this option will disable the output of BLE Mesh debug log.

Default value:
- No (disabled) if *CONFIG_BLE_MESH* && *CONFIG_BLE_MESH*

BLE Mesh STACK DEBUG LOG LEVEL

Contains:

- *CONFIG_BLE_MESH_STACK_TRACE_LEVEL*

CONFIG_BLE_MESH_STACK_TRACE_LEVEL

BLE_MESH_STACK

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh STACK DEBUG LOG LEVEL

Define BLE Mesh trace level for BLE Mesh stack.

Available options:
- NONE (BLE_MESH_TRACE_LEVEL_NONE)
- ERROR (BLE_MESH_TRACE_LEVEL_ERROR)
- WARNING (BLE_MESH_TRACE_LEVEL_WARNING)
- INFO (BLE_MESH_TRACE_LEVEL_INFO)
- DEBUG (BLE_MESH_TRACE_LEVEL_DEBUG)
- VERBOSE (BLE_MESH_TRACE_LEVEL_VERBOSE)

BLE Mesh NET BUF DEBUG LOG LEVEL

Contains:

- *CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL*

CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL

BLE_MESH_NET_BUF

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh NET BUF DEBUG LOG LEVEL

Define BLE Mesh trace level for BLE Mesh net buffer.

Available options:
- NONE (BLE_MESH_NET_BUF_TRACE_LEVEL_NONE)
- ERROR (BLE_MESH_NET_BUF_TRACE_LEVEL_ERROR)
- WARNING (BLE_MESH_NET_BUF_TRACE_LEVEL_WARNING)
- INFO (BLE_MESH_NET_BUF_TRACE_LEVEL_INFO)
- DEBUG (BLE_MESH_NET_BUF_TRACE_LEVEL_DEBUG)
- VERBOSE (BLE_MESH_NET_BUF_TRACE_LEVEL_VERBOSE)

CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT

Timeout(ms) for client message response

Found in: Component config > CONFIG_BLE_MESH

Timeout value used by the node to get response of the acknowledged message which is sent by the client model. This value indicates the maximum time that a client model waits for the response of the sent acknowledged messages. If a client model uses 0 as the timeout value when sending acknowledged messages, then the default value will be used which is four seconds.

Range:
- from 100 to 1200000 if *CONFIG_BLE_MESH*

Default value:
- 4000 if *CONFIG_BLE_MESH*

Support for BLE Mesh Foundation models

Contains:

- *CONFIG_BLE_MESH_CFG_CLI*
- *CONFIG_BLE_MESH_HEALTH_CLI*
- *CONFIG_BLE_MESH_HEALTH_SRV*

CONFIG_BLE_MESH_CFG_CLI

Configuration Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Configuration Client model.

CONFIG_BLE_MESH_HEALTH_CLI

Health Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Health Client model.

CONFIG_BLE_MESH_HEALTH_SRV

Health Server model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Health Server model.

Default value:
- Yes (enabled) if *CONFIG_BLE_MESH*

Support for BLE Mesh Client/Server models

Contains:

- *CONFIG_BLE_MESH_GENERIC_BATTERY_CLI*
- *CONFIG_BLE_MESH_GENERIC_DEF_TRANS_TIME_CLI*
- *CONFIG_BLE_MESH_GENERIC_LEVEL_CLI*
- *CONFIG_BLE_MESH_GENERIC_LOCATION_CLI*
- *CONFIG_BLE_MESH_GENERIC_ONOFF_CLI*
- *CONFIG_BLE_MESH_GENERIC_POWER_LEVEL_CLI*
- *CONFIG_BLE_MESH_GENERIC_POWER_ONOFF_CLI*
- *CONFIG_BLE_MESH_GENERIC_PROPERTY_CLI*
- *CONFIG_BLE_MESH_GENERIC_SERVER*
- *CONFIG_BLE_MESH_LIGHT_CTL_CLI*
- *CONFIG_BLE_MESH_LIGHT_HSL_CLI*
- *CONFIG_BLE_MESH_LIGHT_LC_CLI*

- *CONFIG_BLE_MESH_LIGHT_LIGHTNESS_CLI*
- *CONFIG_BLE_MESH_LIGHT_XYL_CLI*
- *CONFIG_BLE_MESH_LIGHTING_SERVER*
- *CONFIG_BLE_MESH_SCENE_CLI*
- *CONFIG_BLE_MESH_SCHEDULER_CLI*
- *CONFIG_BLE_MESH_SENSOR_CLI*
- *CONFIG_BLE_MESH_SENSOR_SERVER*
- *CONFIG_BLE_MESH_TIME_SCENE_SERVER*
- *CONFIG_BLE_MESH_TIME_CLI*

CONFIG_BLE_MESH_GENERIC_ONOFF_CLI

Generic OnOff Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic OnOff Client model.

CONFIG_BLE_MESH_GENERIC_LEVEL_CLI

Generic Level Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Level Client model.

CONFIG_BLE_MESH_GENERIC_DEF_TRANS_TIME_CLI

Generic Default Transition Time Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Default Transition Time Client model.

CONFIG_BLE_MESH_GENERIC_POWER_ONOFF_CLI

Generic Power OnOff Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Power OnOff Client model.

CONFIG_BLE_MESH_GENERIC_POWER_LEVEL_CLI

Generic Power Level Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Power Level Client model.

CONFIG_BLE_MESH_GENERIC_BATTERY_CLI

Generic Battery Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Battery Client model.

CONFIG_BLE_MESH_GENERIC_LOCATION_CLI

Generic Location Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Location Client model.

CONFIG_BLE_MESH_GENERIC_PROPERTY_CLI

Generic Property Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Property Client model.

CONFIG_BLE_MESH_SENSOR_CLI

Sensor Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Sensor Client model.

CONFIG_BLE_MESH_TIME_CLI

Time Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Time Client model.

CONFIG_BLE_MESH_SCENE_CLI

Scene Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Scene Client model.

CONFIG_BLE_MESH_SCHEDULER_CLI

Scheduler Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Scheduler Client model.

CONFIG_BLE_MESH_LIGHT_LIGHTNESS_CLI

Light Lightness Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Light Lightness Client model.

CONFIG_BLE_MESH_LIGHT_CTL_CLI

Light CTL Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Light CTL Client model.

CONFIG_BLE_MESH_LIGHT_HSL_CLI

Light HSL Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Light HSL Client model.

CONFIG_BLE_MESH_LIGHT_XYL_CLI

Light XYL Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Light XYL Client model.

CONFIG_BLE_MESH_LIGHT_LC_CLI

Light LC Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Light LC Client model.

CONFIG_BLE_MESH_GENERIC_SERVER

Generic server models

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic server models.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SENSOR_SERVER

Sensor server models

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Sensor server models.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_TIME_SCENE_SERVER

Time and Scenes server models

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Time and Scenes server models.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_LIGHTING_SERVER

Lighting server models

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Lighting server models.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_IV_UPDATE_TEST

Test the IV Update Procedure

Found in: Component config > CONFIG_BLE_MESH

This option removes the 96 hour limit of the IV Update Procedure and lets the state to be changed at any time. If IV Update test mode is going to be used, this option should be enabled.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

BLE Mesh specific test option Contains:

- *CONFIG_BLE_MESH_DEBUG*
- *CONFIG_BLE_MESH_SHELL*
- *CONFIG_BLE_MESH_BQB_TEST*
- *CONFIG_BLE_MESH_SELF_TEST*
- *CONFIG_BLE_MESH_TEST_AUTO_ENTER_NETWORK*
- *CONFIG_BLE_MESH_TEST_USE_WHITE_LIST*

CONFIG_BLE_MESH_SELF_TEST

Perform BLE Mesh self-tests

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

This option adds extra self-tests which are run every time BLE Mesh networking is initialized.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_BQB_TEST

Enable BLE Mesh specific internal test

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

This option is used to enable some internal functions for auto-pts test.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_TEST_AUTO_ENTER_NETWORK

Unprovisioned device enters mesh network automatically

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

With this option enabled, an unprovisioned device can automatically enters mesh network using a specific test function without the pro- visioning procedure. And on the Provisioner side, a test function needs to be invoked to add the node information into the mesh stack.

Default value:

- Yes (enabled) if *CONFIG_BLE_MESH_SELF_TEST* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_TEST_USE_WHITE_LIST

Use white list to filter mesh advertising packets

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

With this option enabled, users can use white list to filter mesh advertising packets while scanning.

Default value:

- No (disabled) if *CONFIG_BLE_MESH_SELF_TEST* && *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_SHELL

Enable BLE Mesh shell

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

Activate shell module that provides BLE Mesh commands to the console.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_DEBUG

Enable BLE Mesh debug logs

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

Enable debug logs for the BLE Mesh functionality.

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

CONFIG_BLE_MESH_DEBUG_NET

Network layer debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Network layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_TRANS

Transport layer debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Transport layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_BEACON

Beacon debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Beacon-related debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_CRYPTO

Crypto debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable cryptographic debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_PROV

Provisioning debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Provisioning debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_ACCESS

Access layer debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Access layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_MODEL

Foundation model debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Foundation Models debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_ADV

Advertising debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable advertising debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_LOW_POWER

Low Power debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Low Power debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_FRIEND

Friend debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Friend debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_PROXY

Proxy debug

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_DEBUG

Enable Proxy protocol debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_EXPERIMENTAL

Make BLE Mesh experimental features visible

Found in: Component config > CONFIG_BLE_MESH

Make BLE Mesh Experimental features visible. Experimental features list: - CONFIG_BLE_MESH_NOT_RELAY_REPLAY_MSG

Default value:

- No (disabled) if *CONFIG_BLE_MESH*

Driver Configurations Contains:

- *GPIO Configuration*
- *GPTimer Configuration*
- *I2S Configuration*
- *Legacy ADC Configuration*
- *MCPWM Configuration*
- *PCNT Configuration*
- *RMT Configuration*
- *Sigma Delta Modulator Configuration*
- *SPI Configuration*
- *Temperature sensor Configuration*
- *TWAI Configuration*
- *UART Configuration*

Legacy ADC Configuration Contains:

- *CONFIG_ADC_DISABLE_DAC*
- *Legacy ADC Calibration Configuration*
- *CONFIG_ADC_SUPPRESS_DEPRECATE_WARN*

CONFIG_ADC_DISABLE_DAC

Disable DAC when ADC2 is used on GPIO 25 and 26

Found in: Component config > Driver Configurations > Legacy ADC Configuration

If this is set, the ADC2 driver will disable the output of the DAC corresponding to the specified channel. This is the default value.

For testing, disable this option so that we can measure the output of DAC by internal ADC.

Default value:

- Yes (enabled)

CONFIG_ADC_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > Legacy ADC Configuration

Wether to suppress the deprecation warnings when using legacy adc driver (driver/adc.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

Legacy ADC Calibration Configuration Contains:

- *CONFIG_ADC_CALI_SUPPRESS_DEPRECATE_WARN*
- *CONFIG_ADC_CAL_EFUSE_VREF_ENABLE*
- *CONFIG_ADC_CAL_LUT_ENABLE*
- *CONFIG_ADC_CAL_EFUSE_TP_ENABLE*

CONFIG_ADC_CAL_EFUSE_TP_ENABLE

Use Two Point Values

Found in: Component config > Driver Configurations > Legacy ADC Configuration > Legacy ADC Calibration Configuration

Some ESP32s have Two Point calibration values burned into eFuse BLOCK3. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using Two Point values if they are available.

Default value:

- Yes (enabled)

CONFIG_ADC_CAL_EFUSE_VREF_ENABLE

Use eFuse Vref

Found in: Component config > Driver Configurations > Legacy ADC Configuration > Legacy ADC Calibration Configuration

Some ESP32s have Vref burned into eFuse BLOCK0. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using eFuse Vref if it is available.

Default value:

- Yes (enabled)

CONFIG_ADC_CAL_LUT_ENABLE

Use Lookup Tables

Found in: Component config > Driver Configurations > Legacy ADC Configuration > Legacy ADC Calibration Configuration

This option will allow the ADC calibration component to use Lookup Tables to correct for non-linear behavior in 11db attenuation. Other attenuations do not exhibit non-linear behavior hence will not be affected by this option.

Default value:

- Yes (enabled)

CONFIG_ADC_CALI_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > Legacy ADC Configuration > Legacy ADC Calibration Configuration

Wether to suppress the deprecation warnings when using legacy adc calibration driver (esp_adc_cal.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

SPI Configuration Contains:

- *CONFIG_SPI_MASTER_ISR_IN_IRAM*
- *CONFIG_SPI_SLAVE_ISR_IN_IRAM*
- *CONFIG_SPI_MASTER_IN_IRAM*
- *CONFIG_SPI_SLAVE_IN_IRAM*

CONFIG_SPI_MASTER_IN_IRAM

Place transmitting functions of SPI master into IRAM

Found in: Component config > Driver Configurations > SPI Configuration

Normally only the ISR of SPI master is placed in the IRAM, so that it can work without the flash when interrupt is triggered. For other functions, there's some possibility that the flash cache miss when running inside and out of SPI functions, which may increase the interval of SPI transactions. Enable this to put `queue\_trans`, `get\_trans\_result` and `transmit` functions into the IRAM to avoid possible cache miss.

During unit test, this is enabled to measure the ideal case of api.

Default value:

- No (disabled)

CONFIG_SPI_MASTER_ISR_IN_IRAM

Place SPI master ISR function into IRAM

Found in: Component config > Driver Configurations > SPI Configuration

Place the SPI master ISR in to IRAM to avoid possible cache miss.

Also you can forbid the ISR being disabled during flash writing access, by add ESP_INTR_FLAG_IRAM when initializing the driver.

Default value:

- Yes (enabled)

CONFIG_SPI_SLAVE_IN_IRAM

Place transmitting functions of SPI slave into IRAM

Found in: Component config > Driver Configurations > SPI Configuration

Normally only the ISR of SPI slave is placed in the IRAM, so that it can work without the flash when interrupt is triggered. For other functions, there's some possibility that the flash cache miss when running inside and out of SPI functions, which may increase the interval of SPI transactions. Enable this to put `queue\_trans`, `get\_trans\_result` and `transmit` functions into the IRAM to avoid possible cache miss.

Default value:

- No (disabled)

CONFIG_SPI_SLAVE_ISR_IN_IRAM

Place SPI slave ISR function into IRAM

Found in: Component config > Driver Configurations > SPI Configuration

Place the SPI slave ISR in to IRAM to avoid possible cache miss.

Also you can forbid the ISR being disabled during flash writing access, by add ESP_INTR_FLAG_IRAM when initializing the driver.

Default value:

- Yes (enabled)

TWAI Configuration Contains:

- *CONFIG_TWAI_ERRATA_FIX_RX_FRAME_INVALID*
- *CONFIG_TWAI_ERRATA_FIX_LISTEN_ONLY_DOM*
- *CONFIG_TWAI_ERRATA_FIX_BUS_OFF_REC*
- *CONFIG_TWAI_ERRATA_FIX_RX_FIFO_CORRUPT*
- *CONFIG_TWAI_ERRATA_FIX_TX_INTR_LOST*
- *CONFIG_TWAI_ISR_IN_IRAM*

CONFIG_TWAI_ISR_IN_IRAM

Place TWAI ISR function into IRAM

Found in: Component config > Driver Configurations > TWAI Configuration

Place the TWAI ISR in to IRAM. This will allow the ISR to avoid cache misses, and also be able to run whilst the cache is disabled (such as when writing to SPI Flash). Note that if this option is enabled: - Users should also set the ESP_INTR_FLAG_IRAM in the driver configuration structure when installing the driver (see docs for specifics). - Alert logging (i.e., setting of the TWAI_ALERT_AND_LOG flag) will have no effect.

Default value:

- No (disabled)

CONFIG_TWAI_ERRATA_FIX_BUS_OFF_REC

Add SW workaround for REC change during bus-off

Found in: Component config > Driver Configurations > TWAI Configuration

When the bus-off condition is reached, the REC should be reset to 0 and frozen (via LOM) by the driver's ISR. However on the ESP32, there is an edge case where the REC will increase before the driver's ISR can respond in time (e.g., due to the rapid occurrence of bus errors), thus causing the REC to be non-zero after bus-off. A non-zero REC can prevent bus-off recovery as the bus-off recovery condition is that both TEC and REC become 0. Enabling this option will add a workaround in the driver to forcibly reset REC to zero on reaching bus-off.

Default value:

- Yes (enabled)

CONFIG_TWAI_ERRATA_FIX_TX_INTR_LOST

Add SW workaround for TX interrupt lost errata

Found in: Component config > Driver Configurations > TWAI Configuration

On the ESP32, when a transmit interrupt occurs, and interrupt register is read on the same APB clock cycle, the transmit interrupt could be lost. Enabling this option will add a workaround that checks the transmit buffer status bit to recover any lost transmit interrupt.

Default value:

- Yes (enabled)

CONFIG_TWAI_ERRATA_FIX_RX_FRAME_INVALID

Add SW workaround for invalid RX frame errata

Found in: Component config > Driver Configurations > TWAI Configuration

On the ESP32, when receiving a data or remote frame, if a bus error occurs in the data or CRC field, the data of the next received frame could be invalid. Enabling this option will add a workaround that

will reset the peripheral on detection of this errata condition. Note that if a frame is transmitted on the bus whilst the reset is ongoing, the message will not be receive by the peripheral sent on the bus during the reset, the message will be lost.

Default value:

- Yes (enabled)

CONFIG_TWAI_ERRATA_FIX_RX_FIFO_CORRUPT

Add SW workaround for RX FIFO corruption errata

Found in: Component config > Driver Configurations > TWAI Configuration

On the ESP32, when the RX FIFO overruns and the RX message counter maxes out at 64 messages, the entire RX FIFO is no longer recoverable. Enabling this option will add a workaround that resets the peripheral on detection of this errata condition. Note that if a frame is being sent on the bus during the reset bus during the reset, the message will be lost.

Default value:

- Yes (enabled)

CONFIG_TWAI_ERRATA_FIX_LISTEN_ONLY_DOM

Add SW workaround for listen only transmits dominant bit errata

Found in: Component config > Driver Configurations > TWAI Configuration

When in the listen only mode, the TWAI controller must not influence the TWAI bus (i.e., must not send any dominant bits). However, while in listen only mode on the ESP32/ESP32-S2/ESP32-S3/ESP32-C3, the TWAI controller will still transmit dominant bits when it detects an error (i.e., as part of an active error frame). Enabling this option will add a workaround that forces the TWAI controller into an error passive state on initialization, thus preventing any dominant bits from being sent.

Default value:

- Yes (enabled)

Temperature sensor Configuration

Contains:

- *CONFIG_TEMP_SENSOR_ENABLE_DEBUG_LOG*
- *CONFIG_TEMP_SENSOR_SUPPRESS_DEPRECATE_WARN*

CONFIG_TEMP_SENSOR_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > Temperature sensor Configuration

Wether to suppress the deprecation warnings when using legacy temperature sensor driver (driver/temp_sensor.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled) if SOC_TEMP_SENSOR_SUPPORTED

CONFIG_TEMP_SENSOR_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > Temperature sensor Configuration

Wether to enable the debug log message for temperature sensor driver. Note that, this option only controls the temperature sensor driver log, won't affect other drivers.

Default value:

- No (disabled) if SOC_TEMP_SENSOR_SUPPORTED

UART Configuration Contains:

- *CONFIG_UART_ISR_IN_IRAM*

CONFIG_UART_ISR_IN_IRAM

Place UART ISR function into IRAM

Found in: Component config > Driver Configurations > UART Configuration

If this option is not selected, UART interrupt will be disabled for a long time and may cause data lost when doing spi flash operation.

Default value:

- No (disabled) if *CONFIG_RINGBUF_PLACE_ISR_FUNCTIONS_INTO_FLASH*

GPIO Configuration Contains:

- *CONFIG_GPIO_CTRL_FUNC_IN_IRAM*
- *CONFIG_GPIO_ESP32_SUPPORT_SWITCH_SLP_PULL*

CONFIG_GPIO_ESP32_SUPPORT_SWITCH_SLP_PULL

Support light sleep GPIO pullup/pulldown configuration for ESP32

Found in: Component config > Driver Configurations > GPIO Configuration

This option is intended to fix the bug that ESP32 is not able to switch to configured pullup/pulldown mode in sleep. If this option is selected, chip will automatically emulate the behaviour of switching, and about 450B of source codes would be placed into IRAM.

CONFIG_GPIO_CTRL_FUNC_IN_IRAM

Place GPIO control functions into IRAM

Found in: Component config > Driver Configurations > GPIO Configuration

Place GPIO control functions (like intr_disable/set_level) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context.

Default value:

- No (disabled)

Sigma Delta Modulator Configuration Contains:

- *CONFIG_SDM_ENABLE_DEBUG_LOG*
- *CONFIG_SDM_CTRL_FUNC_IN_IRAM*
- *CONFIG_SDM_SUPPRESS_DEPRECATE_WARN*

CONFIG_SDM_CTRL_FUNC_IN_IRAM

Place SDM control functions into IRAM

Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration

Place SDM control functions (like set_duty) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

Default value:

- No (disabled)

CONFIG_SDM_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration

Wether to suppress the deprecation warnings when using legacy sigma delta driver. If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

CONFIG_SDM_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration

Wether to enable the debug log message for SDM driver. Note that, this option only controls the SDM driver log, won't affect other drivers.

Default value:

- No (disabled)

GPTimer Configuration

Contains:

- *CONFIG_GPTIMER_ENABLE_DEBUG_LOG*
- *CONFIG_GPTIMER_ISR_IRAM_SAFE*
- *CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM*
- *CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN*

CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM

Place GPTimer control functions into IRAM

Found in: Component config > Driver Configurations > GPTimer Configuration

Place GPTimer control functions (like start/stop) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

Default value:

- No (disabled)

CONFIG_GPTIMER_ISR_IRAM_SAFE

GPTimer ISR IRAM-Safe

Found in: Component config > Driver Configurations > GPTimer Configuration

Ensure the GPTimer interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

Default value:

- No (disabled)

CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > GPTimer Configuration

Wether to suppress the deprecation warnings when using legacy timer group driver (driver/timer.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

CONFIG_GPTIMER_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > GPTimer Configuration

Wether to enable the debug log message for GPTimer driver. Note that, this option only controls the GPTimer driver log, won't affect other drivers.

Default value:

- No (disabled)

PCNT Configuration

Contains:

- *CONFIG_PCNT_ENABLE_DEBUG_LOG*
- *CONFIG_PCNT_ISR_IRAM_SAFE*
- *CONFIG_PCNT_CTRL_FUNC_IN_IRAM*
- *CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN*

CONFIG_PCNT_CTRL_FUNC_IN_IRAM

Place PCNT control functions into IRAM

Found in: Component config > Driver Configurations > PCNT Configuration

Place PCNT control functions (like start/stop) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

Default value:

- No (disabled)

CONFIG_PCNT_ISR_IRAM_SAFE

PCNT ISR IRAM-Safe

Found in: Component config > Driver Configurations > PCNT Configuration

Ensure the PCNT interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

Default value:

- No (disabled)

CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > PCNT Configuration

Wether to suppress the deprecation warnings when using legacy PCNT driver (driver/pcnt.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

CONFIG_PCNT_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > PCNT Configuration

Wether to enable the debug log message for PCNT driver. Note that, this option only controls the PCNT driver log, won't affect other drivers.

Default value:

- No (disabled)

RMT Configuration Contains:

- *CONFIG_RMT_ENABLE_DEBUG_LOG*
- *CONFIG_RMT_RECV_FUNC_IN_IRAM*
- *CONFIG_RMT_ISR_IRAM_SAFE*
- *CONFIG_RMT_SUPPRESS_DEPRECATE_WARN*

CONFIG_RMT_ISR_IRAM_SAFE

RMT ISR IRAM-Safe

Found in: Component config > Driver Configurations > RMT Configuration

Ensure the RMT interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

Default value:

- No (disabled)

CONFIG_RMT_RECV_FUNC_IN_IRAM

Place RMT receive function into IRAM

Found in: Component config > Driver Configurations > RMT Configuration

Place RMT receive function into IRAM, so that the receive function can be IRAM-safe and able to be called when the flash cache is disabled. Enabling this option can improve driver performance as well.

Default value:

- No (disabled)

CONFIG_RMT_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > RMT Configuration

Wether to suppress the deprecation warnings when using legacy rmt driver (driver/rmt.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

CONFIG_RMT_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > RMT Configuration

Wether to enable the debug log message for RMT driver. Note that, this option only controls the RMT driver log, won't affect other drivers.

Default value:

- No (disabled)

MCPWM Configuration Contains:

- *CONFIG_MCPWM_ENABLE_DEBUG_LOG*
- *CONFIG_MCPWM_CTRL_FUNC_IN_IRAM*
- *CONFIG_MCPWM_ISR_IRAM_SAFE*
- *CONFIG_MCPWM_SUPPRESS_DEPRECATE_WARN*

CONFIG_MCPWM_ISR_IRAM_SAFE

Place MCPWM ISR function into IRAM

Found in: Component config > Driver Configurations > MCPWM Configuration

This will ensure the MCPWM interrupt handle is IRAM-Safe, allow to avoid flash cache misses, and also be able to run whilst the cache is disabled. (e.g. SPI Flash write)

Default value:

- No (disabled)

CONFIG_MCPWM_CTRL_FUNC_IN_IRAM

Place MCPWM control functions into IRAM

Found in: Component config > Driver Configurations > MCPWM Configuration

Place MCPWM control functions (like set_compare_value) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

Default value:

- No (disabled)

CONFIG_MCPWM_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

Found in: Component config > Driver Configurations > MCPWM Configuration

Wether to suppress the deprecation warnings when using legacy MCPWM driver (driver/mcpwm.h). If you want to continue using the legacy driver, and don't want to see related deprecation warnings, you can enable this option.

Default value:

- No (disabled)

CONFIG_MCPWM_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > Driver Configurations > MCPWM Configuration

Wether to enable the debug log message for MCPWM driver. Note that, this option only controls the MCPWM driver log, won't affect other drivers.

Default value:

- No (disabled)

I2S Configuration Contains:

- *CONFIG_I2S_ENABLE_DEBUG_LOG*
- *CONFIG_I2S_ISR_IRAM_SAFE*
- *CONFIG_I2S_SUPPRESS_DEPRECATE_WARN*

CONFIG_I2S_ISR_IRAM_SAFE

I2S ISR IRAM-Safe

Found in: Component config > Driver Configurations > I2S Configuration

Ensure the I2S interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

Default value:

- No (disabled)

CONFIG_I2S_SUPPRESS_DEPRECATE_WARN

Suppress leagcy driver deprecated warning

Found in: Component config > Driver Configurations > I2S Configuration

Enable this option will suppress the deprecation warnings of using APIs in legacy I2S driver.

Default value:

- No (disabled)

CONFIG_I2S_ENABLE_DEBUG_LOG

Enable I2S debug log

Found in: Component config > Driver Configurations > I2S Configuration

Wether to enable the debug log message for I2S driver. Note that, this option only controls the I2S driver log, will not affect other drivers.

Default value:

- No (disabled)

eFuse Bit Manager Contains:

- *CONFIG_EFUSE_CODE_SCHEME_SELECTOR*
- *CONFIG_EFUSE_VIRTUAL*
- *CONFIG_EFUSE_CUSTOM_TABLE*

CONFIG_EFUSE_CUSTOM_TABLE

Use custom eFuse table

Found in: Component config > eFuse Bit Manager

Allows to generate a structure for eFuse from the CSV file.

Default value:

- No (disabled)

CONFIG_EFUSE_CUSTOM_TABLE_FILENAME

Custom eFuse CSV file

Found in: Component config > eFuse Bit Manager > CONFIG_EFUSE_CUSTOM_TABLE

Name of the custom eFuse CSV filename. This path is evaluated relative to the project root directory.

Default value:

- "main/esp_efuse_custom_table.csv" if *CONFIG_EFUSE_CUSTOM_TABLE*

CONFIG_EFUSE_VIRTUAL

Simulate eFuse operations in RAM

Found in: Component config > eFuse Bit Manager

If "n" - No virtual mode. All eFuse operations are real and use eFuse registers. If "y" - The virtual mode is enabled and all eFuse operations (read and write) are redirected to RAM instead of eFuse registers, all permanent changes (via eFuse) are disabled. Log output will state changes that would be applied, but they will not be.

During startup, the eFuses are copied into RAM. This mode is useful for fast tests.

Default value:

- No (disabled)

CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH

Keep eFuses in flash

Found in: Component config > eFuse Bit Manager > CONFIG_EFUSE_VIRTUAL

In addition to the "Simulate eFuse operations in RAM" option, this option just adds a feature to keep eFuses after reboots in flash memory. To use this mode the partition_table should have the *efuse* partition. partition.csv: "efuse_em, data, efuse, , 0x2000,"

During startup, the eFuses are copied from flash or, in case if flash is empty, from real eFuse to RAM and then update flash. This mode is useful when need to keep changes after reboot (testing secure_boot and flash_encryption).

CONFIG_EFUSE_CODE_SCHEME_SELECTOR

Coding Scheme Compatibility

Found in: Component config > eFuse Bit Manager

Selector eFuse code scheme.

Available options:

- None Only (EFUSE_CODE_SCHEME_COMPAT_NONE)
- 3/4 and None (EFUSE_CODE_SCHEME_COMPAT_3_4)
- Repeat, 3/4 and None (common table does not support it) (EFUSE_CODE_SCHEME_COMPAT_REPEAT)

ESP-TLS

Contains:

- *CONFIG_ESP_TLS_INSECURE*
- *CONFIG_ESP_TLS_LIBRARY_CHOOSE*
- *CONFIG_ESP_TLS_CLIENT_SESSION_TICKETS*
- *CONFIG_ESP_DEBUG_WOLFSSL*
- *CONFIG_ESP_TLS_SERVER*
- *CONFIG_ESP_TLS_PSK_VERIFICATION*
- *CONFIG_ESP_WOLFSSL_SMALL_CERT_VERIFY*

- *CONFIG_ESP_TLS_USE_DS_PERIPHERAL*
- *CONFIG_ESP_TLS_USE_SECURE_ELEMENT*

CONFIG_ESP_TLS_LIBRARY_CHOOSE

Choose SSL/TLS library for ESP-TLS (See help for more Info)

Found in: Component config > ESP-TLS

The ESP-TLS APIs support multiple backend TLS libraries. Currently mbedTLS and WolfSSL are supported. Different TLS libraries may support different features and have different resource usage. Consult the ESP-TLS documentation in ESP-IDF Programming guide for more details.

Available options:

- mbedTLS (ESP_TLS_USING_MBEDTLS)
- wolfSSL (License info in wolfSSL directory README) (ESP_TLS_USING_WOLFSSL)

CONFIG_ESP_TLS_USE_SECURE_ELEMENT

Use Secure Element (ATECC608A) with ESP-TLS

Found in: Component config > ESP-TLS

Enable use of Secure Element for ESP-TLS, this enables internal support for ATECC608A peripheral on ESPWROOM32SE, which can be used for TLS connection.

CONFIG_ESP_TLS_USE_DS_PERIPHERAL

Use Digital Signature (DS) Peripheral with ESP-TLS

Found in: Component config > ESP-TLS

Enable use of the Digital Signature Peripheral for ESP-TLS.The DS peripheral can only be used when it is appropriately configured for TLS. Consult the ESP-TLS documentation in ESP-IDF Programming Guide for more details.

Default value:

- Yes (enabled) if ESP_TLS_USING_MBEDTLS && SOC_DIG_SIGN_SUPPORTED

CONFIG_ESP_TLS_CLIENT_SESSION_TICKETS

Enable client session tickets

Found in: Component config > ESP-TLS

Enable session ticket support as specified in RFC5077.

CONFIG_ESP_TLS_SERVER

Enable ESP-TLS Server

Found in: Component config > ESP-TLS

Enable support for creating server side SSL/TLS session, available for mbedTLS as well as wolfSSL TLS library.

CONFIG_ESP_TLS_SERVER_SESSION_TICKETS

Enable server session tickets

Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_SERVER

Enable session ticket support as specified in RFC5077

CONFIG_ESP_TLS_SERVER_SESSION_TICKET_TIMEOUT

Server session ticket timeout in seconds

Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_SERVER > CONFIG_ESP_TLS_SERVER_SESSION_TICKETS

Sets the session ticket timeout used in the tls server.

Default value:

- 86400 if *CONFIG_ESP_TLS_SERVER_SESSION_TICKETS*

CONFIG_ESP_TLS_SERVER_CERT_SELECT_HOOK

Certificate selection hook

Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_SERVER

Ability to configure and use a certificate selection callback during server handshake, to select a certificate to present to the client based on the TLS extensions supplied in the client hello (alpn, sni, etc).

CONFIG_ESP_TLS_SERVER_MIN_AUTH_MODE_OPTIONAL

ESP-TLS Server: Set minimum Certificate Verification mode to Optional

Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_SERVER

When this option is enabled, the peer (here, the client) certificate is checked by the server, however the handshake continues even if verification failed. By default, the peer certificate is not checked and ignored by the server.

mbedtls_ssl_get_verify_result() can be called after the handshake is complete to retrieve status of verification.

CONFIG_ESP_TLS_PSK_VERIFICATION

Enable PSK verification

Found in: Component config > ESP-TLS

Enable support for pre shared key ciphers, supported for both mbedTLS as well as wolfSSL TLS library.

CONFIG_ESP_TLS_INSECURE

Allow potentially insecure options

Found in: Component config > ESP-TLS

You can enable some potentially insecure options. These options should only be used for testing pusposes. Only enable these options if you are very sure.

CONFIG_ESP_TLS_SKIP_SERVER_CERT_VERIFY

Skip server certificate verification by default (WARNING: ONLY FOR TESTING PURPOSE, READ HELP)

Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_INSECURE

After enabling this option the esp-tls client will skip the server certificate verification by default. Note that this option will only modify the default behaviour of esp-tls client regarding server cert verification. The default behaviour should only be applicable when no other option regarding the server cert verification is opted in the esp-tls config (e.g. crt_bundle_attach, use_global_ca_store etc.). WARNING : Enabling this option comes with a potential risk of establishing a TLS connection with a server which has a fake identity, provided that the server certificate is not provided either through API or other mechanism like ca_store etc.

CONFIG_ESP_WOLFSSL_SMALL_CERT_VERIFY

Enable SMALL_CERT_VERIFY

Found in: Component config > ESP-TLS

Enables server verification with Intermediate CA cert, does not authenticate full chain of trust upto the root CA cert (After Enabling this option client only needs to have Intermediate CA certificate of the server to authenticate server, root CA cert is not necessary).

Default value:

- Yes (enabled) if ESP_TLS_USING_WOLFSSL

CONFIG_ESP_DEBUG_WOLFSSL

Enable debug logs for wolfSSL

Found in: Component config > ESP-TLS

Enable detailed debug prints for wolfSSL SSL library.

ADC and ADC Calibration Contains:

- *ADC Calibration Configurations*
- *CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE*
- *CONFIG_ADC_DISABLE_DAC_OUTPUT*
- *CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM*

CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM

Place ISR version ADC oneshot mode read function into IRAM

Found in: Component config > ADC and ADC Calibration

Place ISR version ADC oneshot mode read function into IRAM.

Default value:

- No (disabled)

CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE

ADC continuous mode driver ISR IRAM-Safe

Found in: Component config > ADC and ADC Calibration

Ensure the ADC continuous mode ISR is IRAM-Safe. When enabled, the ISR handler will be available when the cache is disabled.

Default value:

- No (disabled)

ADC Calibration Configurations Contains:

- *CONFIG_ADC_CALI_EFUSE_VREF_ENABLE*
- *CONFIG_ADC_CALI_LUT_ENABLE*
- *CONFIG_ADC_CALI_EFUSE_TP_ENABLE*

CONFIG_ADC_CALI_EFUSE_TP_ENABLE

Use Two Point Values

Found in: Component config > ADC and ADC Calibration > ADC Calibration Configurations

Some ESP32s have Two Point calibration values burned into eFuse BLOCK3. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using Two Point values if they are available.

Default value:

- Yes (enabled)

CONFIG_ADC_CALI_EFUSE_VREF_ENABLE

Use eFuse Vref

Found in: Component config > ADC and ADC Calibration > ADC Calibration Configurations

Some ESP32s have Vref burned into eFuse BLOCK0. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using eFuse Vref if it is available.

Default value:

- Yes (enabled)

CONFIG_ADC_CALI_LUT_ENABLE

Use Lookup Tables

Found in: Component config > ADC and ADC Calibration > ADC Calibration Configurations

This option will allow the ADC calibration component to use Lookup Tables to correct for non-linear behavior in 11db attenuation. Other attenuations do not exhibit non-linear behavior hence will not be affected by this option.

Default value:

- Yes (enabled)

CONFIG_ADC_DISABLE_DAC_OUTPUT

Disable DAC when ADC2 is in use

Found in: Component config > ADC and ADC Calibration

By default, this is set. The ADC oneshot driver will disable the output of the corresponding DAC channels: ESP32: IO25 and IO26 ESP32S2: IO17 and IO18

Disable this option so as to measure the output of DAC by internal ADC, for test usage.

Default value:

- Yes (enabled)

Common ESP-related

Contains:

- *CONFIG_ESP_ERR_TO_NAME_LOOKUP*

CONFIG_ESP_ERR_TO_NAME_LOOKUP

Enable lookup of error code strings

Found in: Component config > Common ESP-related

Functions esp_err_to_name() and esp_err_to_name_r() return string representations of error codes from a pre-generated lookup table. This option can be used to turn off the use of the look-up table in order

to save memory but this comes at the price of sacrificing distinguishable (meaningful) output string representations.

Default value:

- Yes (enabled)

Ethernet Contains:

- *CONFIG_ETH_TRANSMIT_MUTEX*
- *CONFIG_ETH_USE_ESP32_EMAC*
- *CONFIG_ETH_USE_OPENETH*
- *CONFIG_ETH_USE_SPI_ETHERNET*

CONFIG_ETH_USE_ESP32_EMAC

Support ESP32 internal EMAC controller

Found in: Component config > Ethernet

ESP32 integrates a 10/100M Ethernet MAC controller.

Default value:

- Yes (enabled)

Contains:

- *CONFIG_ETH_DMA_RX_BUFFER_NUM*
- *CONFIG_ETH_DMA_TX_BUFFER_NUM*
- *CONFIG_ETH_IRAM_OPTIMIZATION*
- *CONFIG_ETH_SOFT_FLOW_CONTROL*
- *CONFIG_ETH_DMA_BUFFER_SIZE*
- *CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0*
- *CONFIG_ETH_PHY_INTERFACE*
- *CONFIG_ETH_RMII_CLK_OUT_GPIO*
- *CONFIG_ETH_RMII_CLK_MODE*

CONFIG_ETH_PHY_INTERFACE

PHY interface

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Select the communication interface between MAC and PHY chip.

Available options:

- Reduced Media Independent Interface (RMII) (ETH_PHY_INTERFACE_RMII)

CONFIG_ETH_RMII_CLK_MODE

RMII clock mode

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Select external or internal RMII clock.

Available options:

- Input RMII clock from external (ETH_RMII_CLK_INPUT)
  MAC will get RMII clock from outside. Note that ESP32 only supports GPIO0 to input the RMII clock.
- Output RMII clock from internal (ETH_RMII_CLK_OUTPUT)
  ESP32 can generate RMII clock by internal APLL. This clock can be routed to the external PHY device. ESP32 supports to route the RMII clock to GPIO0/16/17.

CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0

Output RMII clock from GPIO0 (Experimental!)

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

GPIO0 can be set to output a pre-divided PLL clock (test only!). Enabling this option will configure GPIO0 to output a 50MHz clock. In fact this clock doesn't have directly relationship with EMAC peripheral. Sometimes this clock won't work well with your PHY chip. You might need to add some extra devices after GPIO0 (e.g. inverter). Note that outputting RMII clock on GPIO0 is an experimental practice. If you want the Ethernet to work with WiFi, don't select GPIO0 output mode for stability.

Default value:

- No (disabled) if ETH_RMII_CLK_OUTPUT && *CONFIG_ETH_USE_ESP32_EMAC*

CONFIG_ETH_RMII_CLK_OUT_GPIO

RMII clock GPIO number

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Set the GPIO number to output RMII Clock.

Range:

- from 16 to 17 if *CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0* && ETH_RMII_CLK_OUTPUT && *CONFIG_ETH_USE_ESP32_EMAC*

Default value:

- 17 if *CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0* && ETH_RMII_CLK_OUTPUT && *CONFIG_ETH_USE_ESP32_EMAC*

CONFIG_ETH_DMA_BUFFER_SIZE

Ethernet DMA buffer size (Byte)

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Set the size of each buffer used by Ethernet MAC DMA.

Range:

- from 256 to 1600

Default value:

- 512

CONFIG_ETH_DMA_RX_BUFFER_NUM

Amount of Ethernet DMA Rx buffers

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Number of DMA receive buffers. Each buffer's size is ETH_DMA_BUFFER_SIZE. Larger number of buffers could increase throughput somehow.

Range:

- from 3 to 30

Default value:

- 10

CONFIG_ETH_DMA_TX_BUFFER_NUM

Amount of Ethernet DMA Tx buffers

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Number of DMA transmit buffers. Each buffer's size is ETH_DMA_BUFFER_SIZE. Larger number of buffers could increase throughput somehow.

Range:
- from 3 to 30

Default value:
- 10

CONFIG_ETH_SOFT_FLOW_CONTROL

Enable software flow control

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

Ethernet MAC engine on ESP32 doesn't feature a flow control logic. The MAC driver can perform a software flow control if you enable this option. Note that, if the RX buffer number is small, enabling software flow control will cause obvious performance loss.

Default value:
- No (disabled) if *CONFIG_ETH_DMA_RX_BUFFER_NUM* > 15 && *CONFIG_ETH_USE_ESP32_EMAC*

CONFIG_ETH_IRAM_OPTIMIZATION

Enable IRAM optimization

Found in: Component config > Ethernet > CONFIG_ETH_USE_ESP32_EMAC

If enabled, functions related to RX/TX are placed into IRAM. It can improve Ethernet throughput. If disabled, all functions are placed into FLASH.

Default value:
- No (disabled)

CONFIG_ETH_USE_SPI_ETHERNET

Support SPI to Ethernet Module

Found in: Component config > Ethernet

ESP-IDF can also support some SPI-Ethernet modules.

Default value:
- Yes (enabled)

Contains:

- *CONFIG_ETH_SPI_ETHERNET_DM9051*
- *CONFIG_ETH_SPI_ETHERNET_KSZ8851SNL*
- *CONFIG_ETH_SPI_ETHERNET_W5500*

CONFIG_ETH_SPI_ETHERNET_DM9051

Use DM9051

Found in: Component config > Ethernet > CONFIG_ETH_USE_SPI_ETHERNET

DM9051 is a fast Ethernet controller with an SPI interface. It's also integrated with a 10/100M PHY and MAC. Select this to enable DM9051 driver.

CONFIG_ETH_SPI_ETHERNET_W5500

Use W5500 (MAC RAW)

Found in: Component config > Ethernet > CONFIG_ETH_USE_SPI_ETHERNET

W5500 is a HW TCP/IP embedded Ethernet controller. TCP/IP stack, 10/100 Ethernet MAC and PHY are embedded in a single chip. However the driver in ESP-IDF only enables the RAW MAC mode, making it compatible with the software TCP/IP stack. Say yes to enable W5500 driver.

CONFIG_ETH_SPI_ETHERNET_KSZ8851SNL

Use KSZ8851SNL

Found in: Component config > Ethernet > CONFIG_ETH_USE_SPI_ETHERNET

The KSZ8851SNL is a single-chip Fast Ethernet controller consisting of a 10/100 physical layer transceiver (PHY), a MAC, and a Serial Peripheral Interface (SPI). Select this to enable KSZ8851SNL driver.

CONFIG_ETH_USE_OPENETH

Support OpenCores Ethernet MAC (for use with QEMU)

Found in: Component config > Ethernet

OpenCores Ethernet MAC driver can be used when an ESP-IDF application is executed in QEMU. This driver is not supported when running on a real chip.

Default value:

- No (disabled)

Contains:

- *CONFIG_ETH_OPENETH_DMA_RX_BUFFER_NUM*
- *CONFIG_ETH_OPENETH_DMA_TX_BUFFER_NUM*

CONFIG_ETH_OPENETH_DMA_RX_BUFFER_NUM

Number of Ethernet DMA Rx buffers

Found in: Component config > Ethernet > CONFIG_ETH_USE_OPENETH

Number of DMA receive buffers, each buffer is 1600 bytes.

Range:

- from 1 to 64 if *CONFIG_ETH_USE_OPENETH*

Default value:

- 4 if *CONFIG_ETH_USE_OPENETH*

CONFIG_ETH_OPENETH_DMA_TX_BUFFER_NUM

Number of Ethernet DMA Tx buffers

Found in: Component config > Ethernet > CONFIG_ETH_USE_OPENETH

Number of DMA transmit buffers, each buffer is 1600 bytes.

Range:

- from 1 to 64 if *CONFIG_ETH_USE_OPENETH*

Default value:

- 1 if *CONFIG_ETH_USE_OPENETH*

CONFIG_ETH_TRANSMIT_MUTEX

Enable Transmit Mutex

Found in: Component config > Ethernet

Prevents multiple accesses when Ethernet interface is used as shared resource and multiple functionalities might try to access it at a time.

Default value:

- No (disabled)

Event Loop Library Contains:

- *CONFIG_ESP_EVENT_LOOP_PROFILING*
- *CONFIG_ESP_EVENT_POST_FROM_ISR*

CONFIG_ESP_EVENT_LOOP_PROFILING

Enable event loop profiling

Found in: Component config > Event Loop Library

Enables collections of statistics in the event loop library such as the number of events posted to/recieved by an event loop, number of callbacks involved, number of events dropped to to a full event loop queue, run time of event handlers, and number of times/run time of each event handler.

Default value:

- No (disabled)

CONFIG_ESP_EVENT_POST_FROM_ISR

Support posting events from ISRs

Found in: Component config > Event Loop Library

Enable posting events from interrupt handlers.

Default value:

- Yes (enabled)

CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Support posting events from ISRs placed in IRAM

Found in: Component config > Event Loop Library > CONFIG_ESP_EVENT_POST_FROM_ISR

Enable posting events from interrupt handlers placed in IRAM. Enabling this option places API functions esp_event_post and esp_event_post_to in IRAM.

Default value:

- Yes (enabled)

GDB Stub Contains:

- *CONFIG_ESP_GDBSTUB_SUPPORT_TASKS*

CONFIG_ESP_GDBSTUB_SUPPORT_TASKS

Enable listing FreeRTOS tasks through GDB Stub

Found in: Component config > GDB Stub

If enabled, GDBStub can supply the list of FreeRTOS tasks to GDB. Thread list can be queried from GDB using ‘info threads’ command. Note that if GDB task lists were corrupted, this feature may not work. If GDBStub fails, try disabling this feature.

CONFIG_ESP_GDBSTUB_MAX_TASKS

Maximum number of tasks supported by GDB Stub

Found in: Component config > GDB Stub > CONFIG_ESP_GDBSTUB_SUPPORT_TASKS

Set the number of tasks which GDB Stub will support.

Default value:

- 32 if *CONFIG_ESP_GDBSTUB_SUPPORT_TASKS*

ESP HTTP client Contains:

- *CONFIG_ESP_HTTP_CLIENT_ENABLE_BASIC_AUTH*
- *CONFIG_ESP_HTTP_CLIENT_ENABLE_DIGEST_AUTH*
- *CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS*

CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS

Enable https

Found in: Component config > ESP HTTP client

This option will enable https protocol by linking esp-tls library and initializing SSL transport

Default value:

- Yes (enabled)

CONFIG_ESP_HTTP_CLIENT_ENABLE_BASIC_AUTH

Enable HTTP Basic Authentication

Found in: Component config > ESP HTTP client

This option will enable HTTP Basic Authentication. It is disabled by default as Basic auth uses unencrypted encoding, so it introduces a vulnerability when not using TLS

Default value:

- No (disabled)

CONFIG_ESP_HTTP_CLIENT_ENABLE_DIGEST_AUTH

Enable HTTP Digest Authentication

Found in: Component config > ESP HTTP client

This option will enable HTTP Digest Authentication. It is enabled by default, but use of this configuration is not recommended as the password can be derived from the exchange, so it introduces a vulnerability when not using TLS

Default value:

- No (disabled)

HTTP Server Contains:

- *CONFIG_HTTPD_QUEUE_WORK_BLOCKING*
- *CONFIG_HTTPD_PURGE_BUF_LEN*
- *CONFIG_HTTPD_LOG_PURGE_DATA*
- *CONFIG_HTTPD_MAX_REQ_HDR_LEN*
- *CONFIG_HTTPD_MAX_URI_LEN*
- *CONFIG_HTTPD_ERR_RESP_NO_DELAY*
- *CONFIG_HTTPD_WS_SUPPORT*

CONFIG_HTTPD_MAX_REQ_HDR_LEN

Max HTTP Request Header Length

Found in: Component config > HTTP Server

This sets the maximum supported size of headers section in HTTP request packet to be processed by the server

Default value:

- 512

CONFIG_HTTPD_MAX_URI_LEN

Max HTTP URI Length

Found in: Component config > HTTP Server

This sets the maximum supported size of HTTP request URI to be processed by the server

Default value:

- 512

CONFIG_HTTPD_ERR_RESP_NO_DELAY

Use TCP_NODELAY socket option when sending HTTP error responses

Found in: Component config > HTTP Server

Using TCP_NODEALY socket option ensures that HTTP error response reaches the client before the underlying socket is closed. Please note that turning this off may cause multiple test failures

Default value:

- Yes (enabled)

CONFIG_HTTPD_PURGE_BUF_LEN

Length of temporary buffer for purging data

Found in: Component config > HTTP Server

This sets the size of the temporary buffer used to receive and discard any remaining data that is received from the HTTP client in the request, but not processed as part of the server HTTP request handler.

If the remaining data is larger than the available buffer size, the buffer will be filled in multiple iterations. The buffer should be small enough to fit on the stack, but large enough to avoid excessive iterations.

Default value:

- 32

CONFIG_HTTPD_LOG_PURGE_DATA

Log purged content data at Debug level

Found in: Component config > HTTP Server

Enabling this will log discarded binary HTTP request data at Debug level. For large content data this may not be desirable as it will clutter the log.

Default value:

- No (disabled)

CONFIG_HTTPD_WS_SUPPORT

WebSocket server support

Found in: Component config > HTTP Server

This sets the WebSocket server support.

Default value:

- No (disabled)

CONFIG_HTTPD_QUEUE_WORK_BLOCKING

httpd_queue_work as blocking API

Found in: Component config > HTTP Server

This makes httpd_queue_work() API to wait until a message space is available on UDP control socket. It internally uses a counting semaphore with count set to *LWIP_UDP_RECVMBOX_SIZE* to achieve this. This config will slightly change API behavior to block until message gets delivered on control socket.

ESP HTTPS OTA Contains:

- *CONFIG_ESP_HTTPS_OTA_ALLOW_HTTP*
- *CONFIG_ESP_HTTPS_OTA_DECRYPT_CB*

CONFIG_ESP_HTTPS_OTA_DECRYPT_CB

Provide decryption callback

Found in: Component config > ESP HTTPS OTA

Exposes an additional callback whereby firmware data could be decrypted before being processed by OTA update component. This can help to integrate external encryption related format and removal of such encapsulation layer from firmware image.

Default value:

- No (disabled)

CONFIG_ESP_HTTPS_OTA_ALLOW_HTTP

Allow HTTP for OTA (WARNING: ONLY FOR TESTING PURPOSE, READ HELP)

Found in: Component config > ESP HTTPS OTA

It is highly recommended to keep HTTPS (along with server certificate validation) enabled. Enabling this option comes with potential risk of: - Non-encrypted communication channel with server - Accepting firmware upgrade image from server with fake identity

Default value:

- No (disabled)

ESP HTTPS server Contains:

- *CONFIG_ESP_HTTPS_SERVER_ENABLE*

CONFIG_ESP_HTTPS_SERVER_ENABLE

Enable ESP_HTTPS_SERVER component

Found in: Component config > ESP HTTPS server

Enable ESP HTTPS server component

Hardware Settings Contains:

- *Chip revision*
- *ESP_SLEEP_WORKAROUND*
- *GDMA Configuration*
- *MAC Config*
- *Main XTAL Config*
- *Peripheral Control*
- *RTC Clock Config*
- *Sleep Config*

Chip revision Contains:

- *CONFIG_ESP32_REV_MIN*

CONFIG_ESP32_REV_MIN

Minimum Supported ESP32 Revision

Found in: Component config > Hardware Settings > Chip revision

Required minimum chip revision. ESP-IDF will check for it and reject to boot if the chip revision fails the check. This ensures the chip used will have some modifications (features, or bugfixes).

The complied binary will only support chips above this revision, this will also help to reduce binary size.

Available options:

- Rev v0.0 (ECO0) (ESP32_REV_MIN_0)
- Rev v1.0 (ECO1) (ESP32_REV_MIN_1)
- Rev v1.1 (ECO1.1) (ESP32_REV_MIN_1_1)
- Rev v2.0 (ECO2) (ESP32_REV_MIN_2)
- Rev v3.0 (ECO3) (ESP32_REV_MIN_3)
- Rev v3.1 (ECO4) (ESP32_REV_MIN_3_1)

MAC Config Contains:

- *CONFIG_ESP_MAC_IGNORE_MAC_CRC_ERROR*
- *CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES*

CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES

Number of universally administered (by IEEE) MAC address

Found in: Component config > Hardware Settings > MAC Config

Configure the number of universally administered (by IEEE) MAC addresses. During initialization, MAC addresses for each network interface are generated or derived from a single base MAC address. If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address. If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively. When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

Available options:

- Two (ESP32_UNIVERSAL_MAC_ADDRESSES_TWO)
- Four (ESP32_UNIVERSAL_MAC_ADDRESSES_FOUR)

CONFIG_ESP_MAC_IGNORE_MAC_CRC_ERROR

Ignore MAC CRC error (not recommended)

Found in: Component config > Hardware Settings > MAC Config

If you have an invalid MAC CRC (ESP_ERR_INVALID_CRC) problem and you still want to use this chip, you can enable this option to bypass such an error. This applies to both MAC_FACTORY and CUSTOM_MAC efuses.

Default value:

- No (disabled)

Sleep Config Contains:

- *CONFIG_ESP_SLEEP_GPIO_ENABLE_INTERNAL_RESISTORS*
- *CONFIG_ESP_SLEEP_DEEP_SLEEP_WAKEUP_DELAY*
- *CONFIG_ESP_SLEEP_GPIO_RESET_WORKAROUND*
- *CONFIG_ESP_SLEEP_POWER_DOWN_FLASH*
- *CONFIG_ESP_SLEEP_MSPI_NEED_ALL_IO_PU*
- *CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND*
- *CONFIG_ESP_SLEEP_PSRAM_LEAKAGE_WORKAROUND*

CONFIG_ESP_SLEEP_POWER_DOWN_FLASH

Power down flash in light sleep when there is no SPIRAM

Found in: Component config > Hardware Settings > Sleep Config

If enabled, chip will try to power down flash as part of esp_light_sleep_start(), which costs more time when chip wakes up. Can only be enabled if there is no SPIRAM configured.

This option will power down flash under a strict but relatively safe condition. Also, it is possible to power down flash under a relaxed condition by using esp_sleep_pd_config() to set ESP_PD_DOMAIN_VDDSDIO to ESP_PD_OPTION_OFF. It should be noted that there is a risk in powering down flash, you can refer *ESP-IDF Programming Guide/API Reference/System API/Sleep Modes/Power-down of Flash* for more details.

Default value:

- No (disabled) if *CONFIG_SPIRAM*

CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND

Pull-up Flash CS pin in light sleep

Found in: Component config > Hardware Settings > Sleep Config

All IOs will be set to isolate(floating) state by default during sleep. Since the power supply of SPI Flash is not lost during lightsleep, if its CS pin is recognized as low level(selected state) in the floating state, there will be a large current leakage, and the data in Flash may be corrupted by random signals on other SPI pins. Select this option will set the CS pin of Flash to PULL-UP state during sleep, but this will increase the sleep current about 10 uA. If you are developing with esp32xx modules, you must select this option, but if you are developing with chips, you can also pull up the CS pin of SPI Flash in the external circuit to save power consumption caused by internal pull-up during sleep. (!!! Don't deselect this option if you don't have external SPI Flash CS pin pullups.)

Default value:

- Yes (enabled) if *CONFIG_ESP_SLEEP_POWER_DOWN_FLASH*

CONFIG_ESP_SLEEP_PSRAM_LEAKAGE_WORKAROUND

Pull-up PSRAM CS pin in light sleep

Found in: Component config > Hardware Settings > Sleep Config

All IOs will be set to isolate(floating) state by default during sleep. Since the power supply of PSRAM is not lost during lightsleep, if its CS pin is recognized as low level(selected state) in the floating state, there will be a large current leakage, and the data in PSRAM may be corrupted by random signals on other SPI pins. Select this option will set the CS pin of PSRAM to PULL-UP state during sleep, but this will increase the sleep current about 10 uA. If you are developing with esp32xx modules, you must select this option, but if you are developing with chips, you can also pull up the CS pin of PSRAM in the external circuit to save power consumption caused by internal pull-up during sleep. (!!! Don't deselect this option if you don't have external PSRAM CS pin pullups.)

Default value:

- Yes (enabled) if *CONFIG_SPIRAM*

CONFIG_ESP_SLEEP_MSPI_NEED_ALL_IO_PU

Pull-up all SPI pins in light sleep

Found in: Component config > Hardware Settings > Sleep Config

To reduce leakage current, some types of SPI Flash/RAM only need to pull up the CS pin during light sleep. But there are also some kinds of SPI Flash/RAM that need to pull up all pins. It depends on the SPI Flash/RAM chip used.

CONFIG_ESP_SLEEP_GPIO_RESET_WORKAROUND

light sleep GPIO reset workaround

Found in: Component config > Hardware Settings > Sleep Config

esp32c2, esp32c3 and esp32s3 will reset at wake-up if GPIO is received a small electrostatic pulse during light sleep, with specific condition

- GPIO needs to be configured as input-mode only
- The pin receives a small electrostatic pulse, and reset occurs when the pulse voltage is higher than 6 V

For GPIO set to input mode only, it is not a good practice to leave it open/floating, The hardware design needs to controlled it with determined supply or ground voltage is necessary.

This option provides a software workaround for this issue. Configure to isolate all GPIO pins in sleep state.

CONFIG_ESP_SLEEP_DEEP_SLEEP_WAKEUP_DELAY

Extra delay in deep sleep wake stub (in us)

Found in: Component config > Hardware Settings > Sleep Config

When the chip exits deep sleep, the CPU and the flash chip are powered on at the same time. CPU will run deep sleep stub first, and then proceed to load code from flash. Some flash chips need sufficient time to pass between power on and first read operation. By default, without any extra delay, this time is approximately 900us, although some flash chip types need more than that.

By default extra delay is set to 2000us. When optimizing startup time for applications which require it, this value may be reduced.

If you are seeing "flash read err, 1000" message printed to the console after deep sleep reset, try increasing this value.

Range:

- from 0 to 5000

Default value:

- 2000

CONFIG_ESP_SLEEP_GPIO_ENABLE_INTERNAL_RESISTORS

Allow to enable internal pull-up/downs for the Deep-Sleep wakeup IOs

Found in: Component config > Hardware Settings > Sleep Config

When using rtc gpio wakeup source during deepsleep without external pull-up/downs, you may want to make use of the internal ones.

Default value:

- Yes (enabled)

ESP_SLEEP_WORKAROUND

RTC Clock Config Contains:

- *CONFIG_RTC_EXT_CRYST_ADDIT_CURRENT_METHOD*
- *CONFIG_RTC_XTAL_CAL_RETRY*
- *CONFIG_RTC_CLK_CAL_CYCLES*
- *CONFIG_RTC_CLK_SRC*

CONFIG_RTC_CLK_SRC

RTC clock source

Found in: Component config > Hardware Settings > RTC Clock Config

Choose which clock is used as RTC clock source.

- **"Internal 150kHz oscillator" option provides lowest deep sleep current** consumption, and does not require extra external components. However frequency stability with respect to temperature is poor, so time may drift in deep/light sleep modes.
- **"External 32kHz crystal" provides better frequency stability, at the** expense of slightly higher (1uA) deep sleep current consumption.
- **"External 32kHz oscillator" allows using 32kHz clock generated by an** external circuit. In this case, external clock signal must be connected to 32K_XN pin. Amplitude should be <1.2V in case of sine wave signal, and <1V in case of square wave signal. Common mode voltage should be 0.1 < Vcm < 0.5Vamp, where Vamp is the signal amplitude. Additionally, 1nF capacitor must be connected between 32K_XP pin and ground. 32K_XP pin can not be used as a GPIO in this case.
- **"Internal 8.5MHz oscillator divided by 256" option results in higher** deep sleep current (by 5uA) but has better frequency stability than the internal 150kHz oscillator. It does not require external components.

Available options:

- Internal 150kHz RC oscillator (RTC_CLK_SRC_INT_RC)
- External 32kHz crystal (RTC_CLK_SRC_EXT_CRYS)
- External 32kHz oscillator at 32K_XN pin (RTC_CLK_SRC_EXT_OSC)
- Internal 8.5MHz oscillator, divided by 256 (~33kHz) (RTC_CLK_SRC_INT_8MD256)

CONFIG_RTC_EXT_CRYST_ADDIT_CURRENT_METHOD

Additional current for external 32kHz crystal

Found in: Component config > Hardware Settings > RTC Clock Config

With some 32kHz crystal configurations, the X32N and X32P pins may not have enough drive strength to keep the crystal oscillating. Choose the method to provide additional current from touchpad 9 to the external 32kHz crystal. Note that the deep sleep current is slightly high (4-5uA) and the touchpad and the wakeup sources of both touchpad and ULP are not available in method 1 and method 2.

This problem is fixed in ESP32 ECO 3, so this workaround is not needed. Setting the project configuration to minimum revision ECO3 will disable this option, , allow all wakeup sources, and save some code size.

- "None" option will not provide additional current to external crystal
- **"Method 1" option can't ensure 100% to solve the external 32k crystal start failed** issue, but the touchpad can work in this method.
- **"Method 2" option can solve the external 32k issue, but the touchpad can't work** in this method.

Available options:

- None (RTC_EXT_CRYST_ADDIT_CURRENT_NONE)
- Method 1 (RTC_EXT_CRYST_ADDIT_CURRENT)
- Method 2 (RTC_EXT_CRYST_ADDIT_CURRENT_V2)

CONFIG_RTC_CLK_CAL_CYCLES

Number of cycles for RTC_SLOW_CLK calibration

Found in: Component config > Hardware Settings > RTC Clock Config

When the startup code initializes RTC_SLOW_CLK, it can perform calibration by comparing the RTC_SLOW_CLK frequency with main XTAL frequency. This option sets the number of RTC_SLOW_CLK cycles measured by the calibration routine. Higher numbers increase calibration precision, which may be important for applications which spend a lot of time in deep sleep. Lower numbers reduce startup time.

When this option is set to 0, clock calibration will not be performed at startup, and approximate clock frequencies will be assumed:

- 150000 Hz if internal RC oscillator is used as clock source. For this use value 1024.
- **32768 Hz if the 32k crystal oscillator is used. For this use value 3000 or more.** In case more value will help improve the definition of the launch of the crystal. If the crystal could not start, it will be switched to internal RC.

Range:

- from 0 to 27000 if RTC_CLK_SRC_EXT_CRYS || RTC_CLK_SRC_EXT_OSC || RTC_CLK_SRC_INT_8MD256
- from 0 to 32766

Default value:

- 3000 if RTC_CLK_SRC_EXT_CRYS || RTC_CLK_SRC_EXT_OSC || RTC_CLK_SRC_INT_8MD256
- 1024

CONFIG_RTC_XTAL_CAL_RETRY

Number of attempts to repeat 32k XTAL calibration

Found in: Component config > Hardware Settings > RTC Clock Config

Number of attempts to repeat 32k XTAL calibration before giving up and switching to the internal RC. Increase this option if the 32k crystal oscillator does not start and switches to internal RC.

Default value:

- 1 if RTC_CLK_SRC_EXT_CRYS

Peripheral Control Contains:

- *CONFIG_PERIPH_CTRL_FUNC_IN_IRAM*

CONFIG_PERIPH_CTRL_FUNC_IN_IRAM

Place peripheral control functions into IRAM

Found in: Component config > Hardware Settings > Peripheral Control

Place peripheral control functions (e.g. periph_module_reset) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context.

Default value:

- No (disabled)

GDMA Configuration Contains:

- *CONFIG_GDMA_ISR_IRAM_SAFE*
- *CONFIG_GDMA_CTRL_FUNC_IN_IRAM*

CONFIG_GDMA_CTRL_FUNC_IN_IRAM

Place GDMA control functions into IRAM

Found in: Component config > Hardware Settings > GDMA Configuration

Place GDMA control functions (like start/stop/append/reset) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

Default value:

- No (disabled) if SOC_GDMA_SUPPORTED

CONFIG_GDMA_ISR_IRAM_SAFE

GDMA ISR IRAM-Safe

Found in: Component config > Hardware Settings > GDMA Configuration

This will ensure the GDMA interrupt handler is IRAM-Safe, allow to avoid flash cache misses, and also be able to run whilst the cache is disabled. (e.g. SPI Flash write).

Default value:

- No (disabled) if SOC_GDMA_SUPPORTED

Main XTAL Config Contains:

- *CONFIG_XTAL_FREQ_SEL*

CONFIG_XTAL_FREQ_SEL

Main XTAL frequency

Found in: Component config > Hardware Settings > Main XTAL Config

This option selects the operating frequency of the XTAL (crystal) clock used to drive the ESP target. The selected value MUST reflect the frequency of the given hardware.

Note: The XTAL_FREQ_AUTO option allows the ESP target to automatically estimating XTAL clock ' s operating frequency. However, this feature is only supported on the ESP32. The ESP32 uses the internal 8MHZ as a reference when estimating. Due to the internal oscillator ' s frequency being temperature dependent, usage of the XTAL_FREQ_AUTO is not recommended in applications that operate in high ambient temperatures or use high-temperature qualified chips and modules.

Available options:
- 24 MHz (XTAL_FREQ_24)
- 26 MHz (XTAL_FREQ_26)
- 32 MHz (XTAL_FREQ_32)
- 40 MHz (XTAL_FREQ_40)
- Autodetect (XTAL_FREQ_AUTO)

LCD and Touch Panel Contains:

- *LCD Peripheral Configuration*

LCD Peripheral Configuration Contains:

- *CONFIG_LCD_ENABLE_DEBUG_LOG*
- *CONFIG_LCD_PANEL_IO_FORMAT_BUF_SIZE*
- *CONFIG_LCD_RGB_RESTART_IN_VSYNC*
- *CONFIG_LCD_RGB_ISR_IRAM_SAFE*

CONFIG_LCD_PANEL_IO_FORMAT_BUF_SIZE

LCD panel io format buffer size

Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration

LCD driver allocates an internal buffer to transform the data into a proper format, because of the endian order mismatch. This option is to set the size of the buffer, in bytes.

Default value:
- 32

CONFIG_LCD_ENABLE_DEBUG_LOG

Enable debug log

Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration

Wether to enable the debug log message for LCD driver. Note that, this option only controls the LCD driver log, won't affect other drivers.

Default value:
- No (disabled)

CONFIG_LCD_RGB_ISR_IRAM_SAFE

RGB LCD ISR IRAM-Safe

Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration

Ensure the LCD interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write). If you want the LCD driver to keep flushing the screen even when cache ops disabled, you can enable this option. Note, this will also increase the IRAM usage.

Default value:
- No (disabled) if SOC_LCD_RGB_SUPPORTED

CONFIG_LCD_RGB_RESTART_IN_VSYNC

Restart transmission in VSYNC

Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration

Reset the GDMA channel every VBlank to stop permanent desyncs from happening. Only need to enable it when in your application, the DMA can't deliver data as fast as the LCD consumes it.

Default value:

- No (disabled) if SOC_LCD_RGB_SUPPORTED

ESP NETIF Adapter Contains:

- *CONFIG_ESP_NETIF_BRIDGE_EN*
- *CONFIG_ESP_NETIF_L2_TAP*
- *CONFIG_ESP_NETIF_IP_LOST_TIMER_INTERVAL*
- *CONFIG_ESP_NETIF_USE_TCPIP_STACK_LIB*
- *CONFIG_ESP_NETIF_RECEIVE_REPORT_ERRORS*

CONFIG_ESP_NETIF_IP_LOST_TIMER_INTERVAL

IP Address lost timer interval (seconds)

Found in: Component config > ESP NETIF Adapter

The value of 0 indicates the IP lost timer is disabled, otherwise the timer is enabled.

The IP address may be lost because of some reasons, e.g. when the station disconnects from soft-AP, or when DHCP IP renew fails etc. If the IP lost timer is enabled, it will be started everytime the IP is lost. Event SYSTEM_EVENT_STA_LOST_IP will be raised if the timer expires. The IP lost timer is stopped if the station get the IP again before the timer expires.

Range:

- from 0 to 65535

Default value:

- 120

CONFIG_ESP_NETIF_USE_TCPIP_STACK_LIB

TCP/IP Stack Library

Found in: Component config > ESP NETIF Adapter

Choose the TCP/IP Stack to work, for example, LwIP, uIP, etc.

Available options:

- LwIP (ESP_NETIF_TCPIP_LWIP)
  lwIP is a small independent implementation of the TCP/IP protocol suite.
- Loopback (ESP_NETIF_LOOPBACK)
  Dummy implementation of esp-netif functionality which connects driver transmit to receive function. This option is for testing purpose only

CONFIG_ESP_NETIF_RECEIVE_REPORT_ERRORS

Use esp_err_t to report errors from esp_netif_receive

Found in: Component config > ESP NETIF Adapter

Enable if esp_netif_receive() should return error code. This is useful to inform upper layers that packet input to TCP/IP stack failed, so the upper layers could implement flow control. This option is disabled by default due to backward compatibility and will be enabled in v6.0 (IDF-7194)

Default value:

- No (disabled)

CONFIG_ESP_NETIF_L2_TAP

Enable netif L2 TAP support

Found in: Component config > ESP NETIF Adapter

A user program can read/write link layer (L2) frames from/to ESP TAP device. The ESP TAP device can be currently associated only with Ethernet physical interfaces.

CONFIG_ESP_NETIF_L2_TAP_MAX_FDS

Maximum number of opened L2 TAP File descriptors

Found in: Component config > ESP NETIF Adapter > CONFIG_ESP_NETIF_L2_TAP

Maximum number of opened File descriptors (FD's) associated with ESP TAP device. ESP TAP FD's take up a certain amount of memory, and allowing fewer FD's to be opened at the same time conserves memory.

Range:
- from 1 to 10 if *CONFIG_ESP_NETIF_L2_TAP*

Default value:
- 5 if *CONFIG_ESP_NETIF_L2_TAP*

CONFIG_ESP_NETIF_L2_TAP_RX_QUEUE_SIZE

Size of L2 TAP Rx queue

Found in: Component config > ESP NETIF Adapter > CONFIG_ESP_NETIF_L2_TAP

Maximum number of frames queued in opened File descriptor. Once the queue is full, the newly arriving frames are dropped until the queue has enough room to accept incoming traffic (Tail Drop queue management).

Range:
- from 1 to 100 if *CONFIG_ESP_NETIF_L2_TAP*

Default value:
- 20 if *CONFIG_ESP_NETIF_L2_TAP*

CONFIG_ESP_NETIF_BRIDGE_EN

Enable LwIP IEEE 802.1D bridge

Found in: Component config > ESP NETIF Adapter

Enable LwIP IEEE 802.1D bridge support in ESP-NETIF. Note that "Number of clients store data in netif" (LWIP_NUM_NETIF_CLIENT_DATA) option needs to be properly configured to be LwIP bridge avaiable!

Default value:
- No (disabled)

PHY

Contains:

- *CONFIG_ESP_PHY_CALIBRATION_MODE*
- *CONFIG_ESP_PHY_IMPROVE_RX_11B*
- *CONFIG_ESP_PHY_MAX_WIFI_TX_POWER*
- *CONFIG_ESP_PHY_REDUCE_TX_POWER*
- *CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE*
- *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE

Store phy calibration data in NVS

Found in: Component config > PHY

If this option is enabled, NVS will be initialized and calibration data will be loaded from there. PHY calibration will be skipped on deep sleep wakeup. If calibration data is not found, full calibration will be performed and stored in NVS. Normally, only partial calibration will be performed. If this option is disabled, full calibration will be performed.

If it’s easy that your board calibrate bad data, choose ‘n’. Two cases for example, you should choose ‘n’: 1.If your board is easy to be booted up with antenna disconnected. 2.Because of your board design, each time when you do calibration, the result are too unstable. If unsure, choose ‘y’.

Default value:

- Yes (enabled)

CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION

Use a partition to store PHY init data

Found in: Component config > PHY

If enabled, PHY init data will be loaded from a partition. When using a custom partition table, make sure that PHY data partition is included (type: ‘data’, subtype: ‘phy’). With default partition tables, this is done automatically. If PHY init data is stored in a partition, it has to be flashed there, otherwise runtime error will occur.

If this option is not enabled, PHY init data will be embedded into the application binary.

If unsure, choose ‘n’.

Default value:

- No (disabled)

Contains:

- *CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID*
- *CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN*

CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID

Reset default PHY init data if invalid

Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION

If enabled, PHY init data will be restored to default if it cannot be verified successfully to avoid endless bootloops.

If unsure, choose ‘n’.

Default value:

- No (disabled) if *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN

Support multiple PHY init data bin

Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION

If enabled, the corresponding PHY init data type can be automatically switched according to the country code. China’s PHY init data bin is used by default. Can be modified by country information in API esp_wifi_set_country(). The priority of switching the PHY init data type is: 1. Country configured by API esp_wifi_set_country() and the parameter policy is WIFI_COUNTRY_POLICY_MANUAL. 2. Country notified by the connected AP. 3. Country configured by API esp_wifi_set_country() and the parameter policy is WIFI_COUNTRY_POLICY_AUTO.

Default value:

- No (disabled) if *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION* && *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN_EMBED

Support embedded multiple phy init data bin to app bin

Found in: Component config > *PHY* > *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION* > *CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN*

If enabled, multiple phy init data bin will embedded into app bin If not enabled, multiple phy init data bin will still leave alone, and need to be flashed by users.

Default value:

- No (disabled) if *CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN* && *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

CONFIG_ESP_PHY_INIT_DATA_ERROR

Terminate operation when PHY init data error

Found in: Component config > *PHY* > *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION* > *CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN*

If enabled, when an error occurs while the PHY init data is updated, the program will terminate and restart. If not enabled, the PHY init data will not be updated when an error occurs.

Default value:

- No (disabled) if *CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN* && *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

CONFIG_ESP_PHY_MAX_WIFI_TX_POWER

Max WiFi TX power (dBm)

Found in: Component config > *PHY*

Set maximum transmit power for WiFi radio. Actual transmit power for high data rates may be lower than this setting.

Range:

- from 10 to 20

Default value:

- 20

CONFIG_ESP_PHY_REDUCE_TX_POWER

Reduce PHY TX power when brownout reset

Found in: Component config > *PHY*

When brownout reset occurs, reduce PHY TX power to keep the code running.

Default value:

- No (disabled)

CONFIG_ESP_PHY_CALIBRATION_MODE

Calibration mode

Found in: Component config > *PHY*

Select PHY calibration mode. During RF initialization, the partial calibration method is used by default for RF calibration. Full calibration takes about 100ms more than partial calibration. If boot duration is not critical, it is suggested to use the full calibration method. No calibration method is only used when the device wakes up from deep sleep.

Available options:

- Calibration partial (ESP_PHY_RF_CAL_PARTIAL)
- Calibration none (ESP_PHY_RF_CAL_NONE)
- Calibration full (ESP_PHY_RF_CAL_FULL)

CONFIG_ESP_PHY_IMPROVE_RX_11B

Improve Wi-Fi receive 11b pkts

Found in: Component config > PHY

This is a workaround to improve Wi-Fi receive 11b pkts for some modules using AC-DC power supply with high interference, enable this option will sacrifice Wi-Fi OFDM receive performance. But to guarantee 11b receive performance serves as a bottom line in this case.

Default value:

- No (disabled) if SOC_PHY_IMPROVE_RX_11B

Power Management Contains:

- *CONFIG_PM_SLP_DISABLE_GPIO*
- *CONFIG_PM_SLP_IRAM_OPT*
- *CONFIG_PM_RTOS_IDLE_OPT*
- *CONFIG_PM_ENABLE*

CONFIG_PM_ENABLE

Support for power management

Found in: Component config > Power Management

If enabled, application is compiled with support for power management. This option has run-time overhead (increased interrupt latency, longer time to enter idle state), and it also reduces accuracy of RTOS ticks and timers used for timekeeping. Enable this option if application uses power management APIs.

Default value:

- No (disabled) if *CONFIG_FREERTOS_SMP*

CONFIG_PM_DFS_INIT_AUTO

Enable dynamic frequency scaling (DFS) at startup

Found in: Component config > Power Management > CONFIG_PM_ENABLE

If enabled, startup code configures dynamic frequency scaling. Max CPU frequency is set to DEFAULT_CPU_FREQ_MHZ setting, min frequency is set to XTAL frequency. If disabled, DFS will not be active until the application configures it using esp_pm_configure function.

Default value:

- No (disabled) if *CONFIG_PM_ENABLE*

CONFIG_PM_PROFILING

Enable profiling counters for PM locks

Found in: Component config > Power Management > CONFIG_PM_ENABLE

If enabled, esp_pm_* functions will keep track of the amount of time each of the power management locks has been held, and esp_pm_dump_locks function will print this information. This feature can be used to analyze which locks are preventing the chip from going into a lower power state, and see what time the chip spends in each power saving mode. This feature does incur some run-time overhead, so should typically be disabled in production builds.

Default value:

- No (disabled) if *CONFIG_PM_ENABLE*

CONFIG_PM_TRACE

Enable debug tracing of PM using GPIOs

Found in: Component config > Power Management > CONFIG_PM_ENABLE

If enabled, some GPIOs will be used to signal events such as RTOS ticks, frequency switching, entry/exit from idle state. Refer to pm_trace.c file for the list of GPIOs. This feature is intended to be used when analyzing/debugging behavior of power management implementation, and should be kept disabled in applications.

Default value:

- No (disabled) if *CONFIG_PM_ENABLE*

CONFIG_PM_SLP_IRAM_OPT

Put lightsleep related codes in internal RAM

Found in: Component config > Power Management

If enabled, about 1.8KB of lightsleep related source code would be in IRAM and chip would sleep longer for 760us at most each time. This feature is intended to be used when lower power consumption is needed while there is enough place in IRAM to place source code.

CONFIG_PM_RTOS_IDLE_OPT

Put RTOS IDLE related codes in internal RAM

Found in: Component config > Power Management

If enabled, about 260B of RTOS_IDLE related source code would be in IRAM and chip would sleep longer for 40us at most each time. This feature is intended to be used when lower power consumption is needed while there is enough place in IRAM to place source code.

CONFIG_PM_SLP_DISABLE_GPIO

Disable all GPIO when chip at sleep

Found in: Component config > Power Management

This feature is intended to disable all GPIO pins at automantic sleep to get a lower power mode. If enabled, chips will disable all GPIO pins at automantic sleep to reduce about 200~300 uA current. If you want to specifically use some pins normally as chip wakes when chip sleeps, you can call ‘gpio_sleep_sel_dis’ to disable this feature on those pins. You can also keep this feature on and call ‘gpio_sleep_set_direction’ and ‘gpio_sleep_set_pull_mode’ to have a different GPIO configuration at sleep. Waring: If you want to enable this option on ESP32, you should enable *GPIO_ESP32_SUPPORT_SWITCH_SLP_PULL* at first, otherwise you will not be able to switch pullup/pulldown mode.

ESP PSRAM Contains:

- *CONFIG_SPIRAM*

CONFIG_SPIRAM

Support for external, SPI-connected RAM

Found in: Component config > ESP PSRAM

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

Default value:

- No (disabled)

SPI RAM config Contains:

- *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY*
- *CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY*
- *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY*
- *CONFIG_SPIRAM_SPIWP_SD3_PIN*
- *CONFIG_SPIRAM_BANKSWITCH_ENABLE*
- *CONFIG_SPIRAM_2T_MODE*
- *CONFIG_SPIRAM_CACHE_WORKAROUND*
- *CONFIG_SPIRAM_BOOT_INIT*
- *CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL*
- *PSRAM clock and cs IO for ESP32-D2WD*
- *PSRAM clock and cs IO for ESP32-DOWD*
- *PSRAM clock and cs IO for ESP32-PICO-D4*
- *CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL*
- *CONFIG_SPIRAM_MEMTEST*
- *CONFIG_SPIRAM_SPEED*
- *CONFIG_SPIRAM_OCCUPY_SPI_HOST*
- *CONFIG_SPIRAM_USE*
- *SPIRAM cache workaround debugging*
- *SPIRAM workaround libraries placement*
- *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*
- *CONFIG_SPIRAM_TYPE*
- *CONFIG_SPIRAM_CUSTOM_SPIWP_SD3_PIN*

CONFIG_SPIRAM_TYPE

Type of SPI RAM chip in use

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Available options:

- Auto-detect (SPIRAM_TYPE_AUTO)
- ESP-PSRAM16 or APS1604 (SPIRAM_TYPE_ESPPSRAM16)
- ESP-PSRAM32 (SPIRAM_TYPE_ESPPSRAM32)
- ESP-PSRAM64 or LY68L6400 (SPIRAM_TYPE_ESPPSRAM64)

CONFIG_SPIRAM_SPEED

Set RAM clock speed

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Select the speed for the SPI RAM chip. If SPI RAM is enabled, we only support three combinations of SPI speed mode we supported now:

1. Flash SPI running at 40Mhz and RAM SPI running at 40Mhz
2. Flash SPI running at 80Mhz and RAM SPI running at 40Mhz
3. Flash SPI running at 80Mhz and RAM SPI running at 80Mhz

Note: If the third mode(80Mhz+80Mhz) is enabled for SPI RAM of type 32MBit, one of the HSPI/VSPI host will be occupied by the system. Which SPI host to use can be selected by the config item SPIRAM_OCCUPY_SPI_HOST. Application code should never touch HSPI/VSPI hardware in this case. The option to select 80MHz will only be visible if the flash SPI speed is also 80MHz. (ESPTOOLPY_FLASHFREQ_80M is true)

Available options:

- 40MHz clock speed (SPIRAM_SPEED_40M)
- 80MHz clock speed (SPIRAM_SPEED_80M)

CONFIG_SPIRAM_BOOT_INIT

Initialize SPI RAM during startup

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

If this is enabled, the SPI RAM will be enabled during initial boot. Unless you have specific requirements, you'll want to leave this enabled so memory allocated during boot-up can also be placed in SPI RAM.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM*

CONFIG_SPIRAM_IGNORE_NOTFOUND

Ignore PSRAM when not found

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > CONFIG_SPIRAM_BOOT_INIT

Normally, if psram initialization is enabled during compile time but not found at runtime, it is seen as an error making the CPU panic. If this is enabled, booting will complete but no PSRAM will be available. If PSRAM failed to initialize, the following configs may be affected and may need to be corrected manually. SPIRAM_TRY_ALLOCATE_WIFI_LWIP will affect some LWIP and WiFi buffer default values and range values. Enable SPIRAM_TRY_ALLOCATE_WIFI_LWIP, ESP_WIFI_AMSDU_TX_ENABLED, ESP_WIFI_CACHE_TX_BUFFER_NUM and use static WiFi Tx buffer may cause potential memory exhaustion issues. Suggest disable SPIRAM_TRY_ALLOCATE_WIFI_LWIP. Suggest disable ESP_WIFI_AMSDU_TX_ENABLED. Suggest disable ESP_WIFI_CACHE_TX_BUFFER_NUM, need clear CONFIG_FEATURE_CACHE_TX_BUF_BIT of config->feature_caps. Suggest change ESP_WIFI_TX_BUFFER from static to dynamic. Also suggest to adjust some buffer numbers to the values used without PSRAM case. Such as, ESP_WIFI_STATIC_TX_BUFFER_NUM, ESP_WIFI_DYNAMIC_TX_BUFFER_NUM.

Default value:

- No (disabled) if *CONFIG_SPIRAM_BOOT_INIT* && *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY* && *CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_USE

SPI RAM access method

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

The SPI RAM can be accessed in multiple methods: by just having it available as an unmanaged memory region in the CPU's memory map, by integrating it in the heap as 'special' memory needing heap_caps_malloc to allocate, or by fully integrating it making malloc() also able to return SPI RAM pointers.

Available options:

- Integrate RAM into memory map (SPIRAM_USE_MEMMAP)

- Make RAM allocatable using heap_caps_malloc(…, MALLOC_CAP_SPIRAM) (SPIRAM_USE_CAPS_ALLOC)
- Make RAM allocatable using malloc() as well (SPIRAM_USE_MALLOC)

CONFIG_SPIRAM_MEMTEST

Run memory test on SPI RAM initialization

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Runs a rudimentary memory test on initialization. Aborts when memory test fails. Disable this for slightly faster startup.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_BOOT_INIT* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL

Maximum malloc() size, in bytes, to always put in internal memory

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

If malloc() is capable of also allocating SPI-connected ram, its allocation strategy will prefer to allocate chunks less than this size in internal memory, while allocations larger than this will be done from external RAM. If allocation from the preferred region fails, an attempt is made to allocate from the non-preferred region instead, so malloc() will not suddenly fail when either internal or external memory is full.

Range:

- from 0 to 131072 if SPIRAM_USE_MALLOC && *CONFIG_SPIRAM*

Default value:

- 16384 if SPIRAM_USE_MALLOC && *CONFIG_SPIRAM*

CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, allocate internal memory

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, try to allocate internal memory then.

Default value:

- No (disabled) if (SPIRAM_USE_CAPS_ALLOC || SPIRAM_USE_MALLOC) && *CONFIG_SPIRAM*

CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL

Reserve this amount of bytes for data that specifically needs to be in DMA or internal memory

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Because the external/internal RAM allocation strategy is not always perfect, it sometimes may happen that the internal memory is entirely filled up. This causes allocations that are specifically done in internal memory, for example the stack for new tasks or memory to service DMA or have memory that's also available when SPI cache is down, to fail. This option reserves a pool specifically for requests like that; the memory in this pool is not given out when a normal malloc() is called.

Set this to 0 to disable this feature.

Note that because FreeRTOS stacks are forced to internal memory, they will also use this memory pool; be sure to keep this in mind when adjusting this value.

Note also that the DMA reserved pool may not be one single contiguous memory region, depending on the configured size and the static memory usage of the app.

Range:

- from 0 to 262144 if SPIRAM_USE_MALLOC && *CONFIG_SPIRAM*

Default value:

- 32768 if SPIRAM_USE_MALLOC && *CONFIG_SPIRAM*

CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY

Allow .bss segment placed in external memory

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

If enabled, variables with EXT_RAM_BSS_ATTR attribute will be placed in SPIRAM instead of internal DRAM. BSS section of *lwip*, *net80211*, *pp*, *bt* libraries will be automatically placed in SPIRAM. BSS sections from other object files and libraries can also be placed in SPIRAM through linker fragment scheme *extram_bss*.

Note that the variables placed in SPIRAM using EXT_RAM_BSS_ATTR will be zero initialized.

Default value:

- No (disabled) if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY

Allow .noinit segment placed in external memory

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

If enabled, noinit variables can be placed in PSRAM using EXT_RAM_NOINIT_ATTR.

Note the values placed into this section will not be initialized at startup and should keep its value after software restart.

Default value:

- No (disabled) if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_WORKAROUND

Enable workaround for bug in SPI RAM cache for Rev1 ESP32s

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Revision 1 of the ESP32 has a bug that can cause a write to PSRAM not to take place in some situations when the cache line needs to be fetched from external RAM and an interrupt occurs. This enables a fix in the compiler (-mfix-esp32-psram-cache-issue) that makes sure the specific code that is vulnerable to this will not be emitted.

This will also not use any bits of newlib that are located in ROM, opting for a version that is compiled with the workaround and located in flash instead.

The workaround is not required for ESP32 revision 3 and above.

Default value:

- Yes (enabled) if (SPIRAM_USE_MEMMAP || SPIRAM_USE_CAPS_ALLOC || SPIRAM_USE_MALLOC) && *CONFIG_SPIRAM*

SPIRAM cache workaround debugging

Contains:

- *CONFIG_SPIRAM_CACHE_WORKAROUND_STRATEGY*

CONFIG_SPIRAM_CACHE_WORKAROUND_STRATEGY

Workaround strategy

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM cache workaround debugging

Select the workaround strategy. Note that the strategy for precompiled libraries (libgcc, newlib, bt, wifi) is not affected by this selection.

Unless you know you need a different strategy, it's suggested you stay with the default MEMW strategy. Note that DUPLDST can interfere with hardware encryption and this will be automatically disabled if this workaround is selected. 'Insert nops' is the workaround that was used in older esp-idf versions. This workaround still can cause faulty data transfers from/to SPI RAM in some situation.

Available options:

- Insert memw after vulnerable instructions (default) (SPIRAM_CACHE_WORKAROUND_STRATEGY_MEMW)
- Duplicate LD/ST for 32-bit, memw for 8/16 bit (SPIRAM_CACHE_WORKAROUND_STRATEGY_DUPLDST)
- Insert nops between vulnerable loads/stores (old strategy, obsolete) (SPIRAM_CACHE_WORKAROUND_STRATEGY_NOPS)

SPIRAM workaround libraries placement Contains:

- *CONFIG_SPIRAM_CACHE_LIBCHAR_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBENV_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBFILE_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBIO_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBJMP_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBMATH_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBMEM_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBMISC_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBNUMPARSER_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBRAND_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBSTR_IN_IRAM*
- *CONFIG_SPIRAM_CACHE_LIBTIME_IN_IRAM*

CONFIG_SPIRAM_CACHE_LIBJMP_IN_IRAM

Put libc's jump related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: longjmp and setjmp. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBMATH_IN_IRAM

Put libc's math related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: abs, div, labs, ldiv, quorem, fpclassify, and nan. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBNUMPARSER_IN_IRAM

Put libc' s number parsing related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: utoa, itoa, atoi, atol, strtol, and strtoul. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBIO_IN_IRAM

Put libc' s I/O related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: wcrtomb, fvwrite, wbuf, wsetup, fputwc, wctomb_r, ungetc, makebuf, fflush, refill, and sccl. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBTIME_IN_IRAM

Put libc' s time related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: asctime, asctime_r, ctime, ctime_r, lcltime, lcltime_r, gmtime, gmtime_r, strftime, mktime, tzset_r, tzset, time, gettzinfo, systimes, month_lengths, timelocal, tzvars, tzlock, tzcalc_limits, and strptime. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBCHAR_IN_IRAM

Put libc' s characters related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: ctype_, toupper, tolower, toascii, strupr, bzero, isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, and isupper. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBMEM_IN_IRAM

Put libc' s memory related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: memccpy, memchr memmove, and memrchr. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBSTR_IN_IRAM

Put libc' s string related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: strcasecmp, strcasestr, strchr, strcoll, strcpy, strcspn, strdup, strdup_r, strlcat, strlcpy, strlen, strlwr, strncasecmp, strncat, strncmp, strncpy, strndup, strndup_r, strrchr, strsep, strspn, strstr, strtok_r, and strupr. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBRAND_IN_IRAM

Put libc' s random related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: srand, rand, and rand_r. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBENV_IN_IRAM

Put libc' s environment related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: environ, envlock, and getenv_r. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBFILE_IN_IRAM

Put libc' s file related functions in IRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > SPIRAM workaround libraries placement

The functions affected by this option are: lock, isatty, fclose, open, close, creat, read, rshift, sbrk, stdio, syssbrk, sysclose, sysopen, creat, sysread, syswrite, impure, fwalk, and findfp. Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CACHE_LIBMISC_IN_IRAM

Put libc' s miscellaneous functions in IRAM, see help

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config* > *SPIRAM workaround libraries placement*

The functions affected by this option are: raise and system Putting these function in IRAM will allow them to be called when flash cache is disabled but it will also reduce the available size of free IRAM for the user application.

Default value:

- Yes (enabled) if *CONFIG_SPIRAM_CACHE_WORKAROUND* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_BANKSWITCH_ENABLE

Enable bank switching for >4MiB external RAM

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config*

The ESP32 only supports 4MiB of external RAM in its address space. The hardware does support larger memories, but these have to be bank-switched in and out of this address space. Enabling this allows you to reserve some MMU pages for this, which allows the use of the esp_himem api to manage these banks.

#Note that this is limited to 62 banks, as esp_psram_extram_writeback_cache needs some kind of mapping of #some banks below that mark to work. We cannot at this moment guarantee this to exist when himem is #enabled.

If spiram 2T mode is enabled, the size of 64Mbit psram will be changed as 32Mbit, so himem will be unusable.

Default value:

- Yes (enabled) if (SPIRAM_USE_MEMMAP || SPIRAM_USE_CAPS_ALLOC || SPIRAM_USE_MALLOC) && *CONFIG_SPIRAM*

CONFIG_SPIRAM_BANKSWITCH_RESERVE

Amount of 32K pages to reserve for bank switching

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config* > *CONFIG_SPIRAM_BANKSWITCH_ENABLE*

Select the amount of banks reserved for bank switching. Note that the amount of RAM allocatable with malloc/esp_heap_alloc_caps will decrease by 32K for each page reserved here.

Note that this reservation is only actually done if your program actually uses the himem API. Without any himem calls, the reservation is not done and the original amount of memory will be available to malloc/esp_heap_alloc_caps.

Range:

- from 1 to 62 if *CONFIG_SPIRAM_BANKSWITCH_ENABLE* && *CONFIG_SPIRAM*

Default value:

- 8 if *CONFIG_SPIRAM_BANKSWITCH_ENABLE* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY

Allow external memory as an argument to xTaskCreateStatic

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

Because some bits of the ESP32 code environment cannot be recompiled with the cache workaround, normally tasks cannot be safely run with their stack residing in external memory; for this reason xTaskCreate (and related task creaton functions) always allocate stack in internal memory and xTaskCreateStatic will check if the memory passed to it is in internal memory. If you have a task that needs a large amount of stack and does not call on ROM code in any way (no direct calls, but also no Bluetooth/WiFi), you can try enable this to cause xTaskCreateStatic to allow tasks stack in external memory.

Default value:

- No (disabled) if SPIRAM_USE_MALLOC && *CONFIG_SPIRAM*

CONFIG_SPIRAM_OCCUPY_SPI_HOST

SPI host to use for 32MBit PSRAM

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config

When both flash and PSRAM is working under 80MHz, and the PSRAM is of type 32MBit, one of the HSPI/VSPI host will be used to output the clock. Select which one to use here.

Available options:

- HSPI host (SPI2) (SPIRAM_OCCUPY_HSPI_HOST)
- VSPI host (SPI3) (SPIRAM_OCCUPY_VSPI_HOST)
- Will not try to use any host, will abort if not able to use the PSRAM (SPIRAM_OCCUPY_NO_HOST)

PSRAM clock and cs IO for ESP32-DOWD

Contains:

- *CONFIG_D0WD_PSRAM_CLK_IO*
- *CONFIG_D0WD_PSRAM_CS_IO*

CONFIG_D0WD_PSRAM_CLK_IO

PSRAM CLK IO number

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > PSRAM clock and cs IO for ESP32-DOWD

The PSRAM CLOCK IO can be any unused GPIO, user can config it based on hardware design. If user use 1.8V flash and 1.8V psram, this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

If configured to the same pin as Flash, PSRAM shouldn't be rev0. Contact Espressif for more information.

Range:

- from 0 to 33 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

Default value:

- 17 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

CONFIG_D0WD_PSRAM_CS_IO

PSRAM CS IO number

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > PSRAM clock and cs IO for ESP32-DOWD

The PSRAM CS IO can be any unused GPIO, user can config it based on hardware design. If user use 1.8V flash and 1.8V psram, this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

Range:
- from 0 to 33 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

Default value:
- 16 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

PSRAM clock and cs IO for ESP32-D2WD Contains:

- *CONFIG_D2WD_PSRAM_CLK_IO*
- *CONFIG_D2WD_PSRAM_CS_IO*

CONFIG_D2WD_PSRAM_CLK_IO

PSRAM CLK IO number

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > PSRAM clock and cs IO for ESP32-D2WD

User can config it based on hardware design. For ESP32-D2WD chip, the psram can only be 1.8V psram, so this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

If configured to the same pin (GPIO6) as Flash, PSRAM shouldn't be rev0. Contact Espressif for more information.

Range:
- from 0 to 33 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

Default value:
- 9 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

CONFIG_D2WD_PSRAM_CS_IO

PSRAM CS IO number

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > PSRAM clock and cs IO for ESP32-D2WD

User can config it based on hardware design. For ESP32-D2WD chip, the psram can only be 1.8V psram, so this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

Range:
- from 0 to 33 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

Default value:
- 10 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

PSRAM clock and cs IO for ESP32-PICO-D4 Contains:

- *CONFIG_PICO_PSRAM_CS_IO*

CONFIG_PICO_PSRAM_CS_IO

PSRAM CS IO number

Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config > PSRAM clock and cs IO for ESP32-PICO-D4

The PSRAM CS IO can be any unused GPIO, user can config it based on hardware design.

For ESP32-PICO chip, the psram share clock with flash, so user do not need to configure the clock IO. For the reference hardware design, please refer to https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf

Range:
- from 0 to 33 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

Default value:

- 10 if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

CONFIG_SPIRAM_CUSTOM_SPIWP_SD3_PIN

Use custom SPI PSRAM WP(SD3) Pin when flash pins set in eFuse (read help)

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config*

This setting is only used if the SPI flash pins have been overridden by setting the eFuses SPI_PAD_CONFIG_xxx, and the SPI flash mode is DIO or DOUT.

When this is the case, the eFuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin "SD_DATA_3" or SPI flash pin "IO2") is not specified in eFuse. The psram only has QPI mode, so a WP pin setting is necessary.

If this config item is set to N (default), the correct WP pin will be automatically used for any Espressif chip or module with integrated flash. If a custom setting is needed, set this config item to Y and specify the GPIO number connected to the WP pin.

When flash mode is set to QIO or QOUT, the PSRAM WP pin will be set the same as the SPI Flash WP pin configured in the bootloader.

Default value:

- No (disabled) if (ESPTOOLPY_FLASHMODE_DIO || ESPTOOLPY_FLASHMODE_DOUT) && *CONFIG_SPIRAM*

CONFIG_SPIRAM_SPIWP_SD3_PIN

Custom SPI PSRAM WP(SD3) Pin

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config*

The option "Use custom SPI PSRAM WP(SD3) pin" must be set or this value is ignored

If burning a customized set of SPI flash pins in eFuse and using DIO or DOUT mode for flash, set this value to the GPIO number of the SPIRAM WP pin.

Range:

- from 0 to 33 if (ESPTOOLPY_FLASHMODE_DIO || ESPTOOLPY_FLASHMODE_DOUT) && *CONFIG_SPIRAM*

Default value:

- 7 if (ESPTOOLPY_FLASHMODE_DIO || ESPTOOLPY_FLASHMODE_DOUT) && *CONFIG_SPIRAM*

CONFIG_SPIRAM_2T_MODE

Enable SPI PSRAM 2T mode

Found in: Component config > *ESP PSRAM* > *CONFIG_SPIRAM* > *SPI RAM config*

Enable this option to fix single bit errors inside 64Mbit PSRAM.

Some 64Mbit PSRAM chips have a hardware issue in the RAM which causes bit errors at multiple fixed bit positions.

Note: If this option is enabled, the 64Mbit PSRAM chip will appear to be 32Mbit in size. Applications will not be affected unless the use the esp_himem APIs, which are not supported in 2T mode.

Default value:

- No (disabled) if *CONFIG_SPIRAM* && *CONFIG_SPIRAM*

ESP Ringbuf Contains:

- *CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH*

CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH

Place non-ISR ringbuf functions into flash

Found in: Component config > ESP Ringbuf

Place non-ISR ringbuf functions (like xRingbufferCreate/xRingbufferSend) into flash. This frees up IRAM, but the functions can no longer be called when the cache is disabled.

Default value:

- No (disabled)

CONFIG_RINGBUF_PLACE_ISR_FUNCTIONS_INTO_FLASH

Place ISR ringbuf functions into flash

Found in: Component config > ESP Ringbuf > CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH

Place ISR ringbuf functions (like xRingbufferSendFromISR/xRingbufferReceiveFromISR) into flash. This frees up IRAM, but the functions can no longer be called when the cache is disabled or from an IRAM interrupt context.

This option is not compatible with ESP-IDF drivers which are configured to run the ISR from an IRAM context, e.g. CONFIG_UART_ISR_IN_IRAM.

Default value:

- No (disabled) if *CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH*

ESP System Settings

Contains:

- *CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES*
- *Brownout Detector*
- *CONFIG_ESP_CONSOLE_UART*
- *CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ*
- *CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP*
- *CONFIG_ESP_TASK_WDT_EN*
- *CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE*
- *CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL*
- *CONFIG_ESP_INT_WDT*
- *CONFIG_ESP_MAIN_TASK_AFFINITY*
- *CONFIG_ESP_MAIN_TASK_STACK_SIZE*
- *CONFIG_ESP_DEBUG_OCDAWARE*
- *Memory*
- *Memory protection*
- *CONFIG_ESP_MINIMAL_SHARED_STACK_SIZE*
- *CONFIG_ESP_DEBUG_STUBS_ENABLE*
- *CONFIG_ESP_SYSTEM_PANIC*
- *CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE*
- *CONFIG_ESP_PANIC_HANDLER_IRAM*
- *CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE*
- *Trace memory*
- *CONFIG_ESP_CONSOLE_UART_BAUDRATE*
- *CONFIG_ESP_CONSOLE_UART_NUM*
- *CONFIG_ESP_CONSOLE_UART_RX_GPIO*
- *CONFIG_ESP_CONSOLE_UART_TX_GPIO*

CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ

CPU frequency

Found in: Component config > ESP System Settings

CPU frequency to be set on application startup.

Available options:

- 40 MHz (ESP_DEFAULT_CPU_FREQ_MHZ_40)
- 80 MHz (ESP_DEFAULT_CPU_FREQ_MHZ_80)
- 160 MHz (ESP_DEFAULT_CPU_FREQ_MHZ_160)
- 240 MHz (ESP_DEFAULT_CPU_FREQ_MHZ_240)

Memory Contains:

- *CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY*
- *CONFIG_ESP32_RTCDATA_IN_FAST_MEM*
- *CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE*

CONFIG_ESP32_RTCDATA_IN_FAST_MEM

Place RTC_DATA_ATTR and RTC_RODATA_ATTR variables into RTC fast memory segment

Found in: Component config > ESP System Settings > Memory

This option allows to place .rtc_data and .rtc_rodata sections into RTC fast memory segment to free the slow memory region for ULP programs. This option depends on the CONFIG_FREERTOS_UNICORE option because RTC fast memory can be accessed only by PRO_CPU core.

Default value:

- No (disabled) if *CONFIG_FREERTOS_UNICORE*

CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE

Use fixed static RAM size

Found in: Component config > ESP System Settings > Memory

If this option is disabled, the DRAM part of the heap starts right after the .bss section, within the dram0_0 region. As a result, adding or removing some static variables will change the available heap size.

If this option is enabled, the DRAM part of the heap starts right after the dram0_0 region, where its length is set with ESP32_FIXED_STATIC_RAM_SIZE

Default value:

- No (disabled)

CONFIG_ESP32_FIXED_STATIC_RAM_SIZE

Fixed Static RAM size

Found in: Component config > ESP System Settings > Memory > CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE

RAM size dedicated for static variables (.data & .bss sections). Please note that the actual length will be reduced by BTDM_RESERVE_DRAM if Bluetooth controller is enabled.

Range:

- from 0 to 0x2c200 if *CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE*

Default value:

- "0x1E000" if *CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE*

CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY

Enable IRAM as 8 bit accessible memory

Found in: Component config > ESP System Settings > Memory

If enabled, application can use IRAM as byte accessible region for storing data (Note: IRAM region cannot be used as task stack)

This is possible due to handling of exceptions *LoadStoreError (3)* and *LoadStoreAlignmentError (9)* Each unaligned read/write access will incur a penalty of maximum of 167 CPU cycles.

Trace memory Contains:

- *CONFIG_ESP32_TRAX*

CONFIG_ESP32_TRAX

Use TRAX tracing feature

Found in: Component config > ESP System Settings > Trace memory

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

Default value:

- No (disabled)

CONFIG_ESP32_TRAX_TWOBANKS

Reserve memory for tracing both pro as well as app cpu execution

Found in: Component config > ESP System Settings > Trace memory > CONFIG_ESP32_TRAX

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

Memory to reverse for trace, used in linker script

Default value:

- No (disabled) if *CONFIG_ESP32_TRAX* && *CONFIG_FREERTOS_UNICORE*

CONFIG_ESP_SYSTEM_PANIC

Panic handler behaviour

Found in: Component config > ESP System Settings

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked. Configure the panic handler's action here.

Available options:

- Print registers and halt (ESP_SYSTEM_PANIC_PRINT_HALT)
  Outputs the relevant registers over the serial port and halt the processor. Needs a manual reset to restart.
- Print registers and reboot (ESP_SYSTEM_PANIC_PRINT_REBOOT)
  Outputs the relevant registers over the serial port and immediately reset the processor.
- Silent reboot (ESP_SYSTEM_PANIC_SILENT_REBOOT)
  Just resets the processor without outputting anything
- GDBStub on panic (ESP_SYSTEM_PANIC_GDBSTUB)
  Invoke gdbstub on the serial port, allowing for gdb to attach to it to do a postmortem of the crash.
- GDBStub at runtime (ESP_SYSTEM_GDBSTUB_RUNTIME)
  Invoke gdbstub on the serial port, allowing for gdb to attach to it and to do a debug on runtime.

CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES

Bootstrap cycles for external 32kHz crystal

Found in: Component config > ESP System Settings

To reduce the startup time of an external RTC crystal, we bootstrap it with a 32kHz square wave for a fixed number of cycles. Setting 0 will disable bootstrapping (if disabled, the crystal may take longer to start up or fail to oscillate under some conditions).

If this value is too high, a faulty crystal may initially start and then fail. If this value is too low, an otherwise good crystal may not start.

To accurately determine if the crystal has started, set a larger "Number of cycles for RTC_SLOW_CLK calibration" (about 3000).

CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP

Enable RTC fast memory for dynamic allocations

Found in: Component config > ESP System Settings

This config option allows to add RTC fast memory region to system heap with capability similar to that of DRAM region but without DMA. This memory will be consumed first per heap initialization order by early startup services and scheduler related code. Speed wise RTC fast memory operates on APB clock and hence does not have much performance impact.

Memory protection Contains:

- *CONFIG_ESP_SYSTEM_PMP_IDRAM_SPLIT*
- *CONFIG_ESP_SYSTEM_MEMPROT_FEATURE*

CONFIG_ESP_SYSTEM_PMP_IDRAM_SPLIT

Enable IRAM/DRAM split protection

Found in: Component config > ESP System Settings > Memory protection

If enabled, the CPU watches all the memory access and raises an exception in case of any memory violation. This feature automatically splits the SRAM memory, using PMP, into data and instruction segments and sets Read/Execute permissions for the instruction part (below given splitting address) and Read/Write permissions for the data part (above the splitting address). The memory protection is effective on all access through the IRAM0 and DRAM0 buses.

Default value:

- Yes (enabled) if SOC_CPU_IDRAM_SPLIT_USING_PMP

CONFIG_ESP_SYSTEM_MEMPROT_FEATURE

Enable memory protection

Found in: Component config > ESP System Settings > Memory protection

If enabled, the permission control module watches all the memory access and fires the panic handler if a permission violation is detected. This feature automatically splits the SRAM memory into data and instruction segments and sets Read/Execute permissions for the instruction part (below given splitting address) and Read/Write permissions for the data part (above the splitting address). The memory protection is effective on all access through the IRAM0 and DRAM0 buses.

Default value:

- Yes (enabled) if SOC_MEMPROT_SUPPORTED

CONFIG_ESP_SYSTEM_MEMPROT_FEATURE_LOCK

Lock memory protection settings

Found in: Component config > ESP System Settings > Memory protection > CONFIG_ESP_SYSTEM_MEMPROT_FEATURE

Once locked, memory protection settings cannot be changed anymore. The lock is reset only on the chip startup.

Default value:

- Yes (enabled) if *CONFIG_ESP_SYSTEM_MEMPROT_FEATURE*

CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE

System event queue size

Found in: Component config > ESP System Settings

Config system event queue size in different application.

Default value:

- 32

CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE

Event loop task stack size

Found in: Component config > ESP System Settings

Config system event task stack size in different application.

Default value:

- 2304

CONFIG_ESP_MAIN_TASK_STACK_SIZE

Main task stack size

Found in: Component config > ESP System Settings

Configure the "main task" stack size. This is the stack of the task which calls app_main(). If app_main() returns then this task is deleted and its stack memory is freed.

Default value:

- 3584

CONFIG_ESP_MAIN_TASK_AFFINITY

Main task core affinity

Found in: Component config > ESP System Settings

Configure the "main task" core affinity. This is the used core of the task which calls app_main(). If app_main() returns then this task is deleted.

Available options:

- CPU0 (ESP_MAIN_TASK_AFFINITY_CPU0)
- CPU1 (ESP_MAIN_TASK_AFFINITY_CPU1)
- No affinity (ESP_MAIN_TASK_AFFINITY_NO_AFFINITY)

CONFIG_ESP_MINIMAL_SHARED_STACK_SIZE

Minimal allowed size for shared stack

Found in: Component config > ESP System Settings

Minimal value of size, in bytes, accepted to execute a expression with shared stack.

Default value:

- 2048

CONFIG_ESP_CONSOLE_UART

Channel for console output

Found in: Component config > ESP System Settings

Select where to send console output (through stdout and stderr).

- Default is to use UART0 on pre-defined GPIOs.
- If "Custom" is selected, UART0 or UART1 can be chosen, and any pins can be selected.
- If "None" is selected, there will be no console output on any UART, except for initial output from ROM bootloader. This ROM output can be suppressed by GPIO strapping or EFUSE, refer to chip datasheet for details.
- On chips with USB OTG peripheral, "USB CDC" option redirects output to the CDC port. This option uses the CDC driver in the chip ROM. This option is incompatible with TinyUSB stack.
- On chips with an USB serial/JTAG debug controller, selecting the option for that redirects output to the CDC/ACM (serial port emulation) component of that device.

Available options:

- Default: UART0 (ESP_CONSOLE_UART_DEFAULT)
- USB CDC (ESP_CONSOLE_USB_CDC)
- USB Serial/JTAG Controller (ESP_CONSOLE_USB_SERIAL_JTAG)
- Custom UART (ESP_CONSOLE_UART_CUSTOM)
- None (ESP_CONSOLE_NONE)

CONFIG_ESP_CONSOLE_UART_NUM

UART peripheral to use for console output (0-1)

Found in: Component config > ESP System Settings

This UART peripheral is used for console output from the ESP-IDF Bootloader and the app.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

Due to an ESP32 ROM bug, UART2 is not supported for console output via esp_rom_printf.

Available options:

- UART0 (ESP_CONSOLE_UART_CUSTOM_NUM_0)
- UART1 (ESP_CONSOLE_UART_CUSTOM_NUM_1)

CONFIG_ESP_CONSOLE_UART_TX_GPIO

UART TX on GPIO#

Found in: Component config > ESP System Settings

This GPIO is used for console UART TX output in the ESP-IDF Bootloader and the app (including boot log output and default standard output and standard error of the app).

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

Range:

- from 0 to 46 if ESP_CONSOLE_UART_CUSTOM

Default value:

- 1 if ESP_CONSOLE_UART_CUSTOM
- 43 if ESP_CONSOLE_UART_CUSTOM

CONFIG_ESP_CONSOLE_UART_RX_GPIO

UART RX on GPIO#

Found in: Component config > ESP System Settings

This GPIO is used for UART RX input in the ESP-IDF Bootloader and the app (including default default standard input of the app).

Note: The default ESP-IDF Bootloader configures this pin but doesn't read anything from the UART.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

Range:

- from 0 to 46 if ESP_CONSOLE_UART_CUSTOM

Default value:

- 3 if ESP_CONSOLE_UART_CUSTOM
- 44 if ESP_CONSOLE_UART_CUSTOM

CONFIG_ESP_CONSOLE_UART_BAUDRATE

UART console baud rate

Found in: Component config > ESP System Settings

This baud rate is used by both the ESP-IDF Bootloader and the app (including boot log output and default standard input/output/error of the app).

The app's maximum baud rate depends on the UART clock source. If Power Management is disabled, the UART clock source is the APB clock and all baud rates in the available range will be sufficiently accurate. If Power Management is enabled, REF_TICK clock source is used so the baud rate is divided from 1MHz. Baud rates above 1Mbps are not possible and values between 500Kbps and 1Mbps may not be accurate.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

Range:

- from 1200 to 4000000 if *CONFIG_PM_ENABLE*
- from 1200 to 1000000 if *CONFIG_PM_ENABLE*

Default value:

- 115200

CONFIG_ESP_INT_WDT

Interrupt watchdog

Found in: Component config > ESP System Settings

This watchdog timer can detect if the FreeRTOS tick interrupt has not been called for a certain time, either because a task turned off interrupts and did not turn them on for a long time, or because an interrupt handler did not return. It will try to invoke the panic handler first and failing that reset the SoC.

Default value:

- Yes (enabled)

CONFIG_ESP_INT_WDT_TIMEOUT_MS

Interrupt watchdog timeout (ms)

Found in: Component config > ESP System Settings > CONFIG_ESP_INT_WDT

The timeout of the watchdog, in miliseconds. Make this higher than the FreeRTOS tick rate.

Range:
- from 10 to 10000

Default value:
- 300 if *CONFIG_SPIRAM* && *CONFIG_ESP_INT_WDT*
- 800 if *CONFIG_SPIRAM* && *CONFIG_ESP_INT_WDT*

CONFIG_ESP_INT_WDT_CHECK_CPU1

Also watch CPU1 tick interrupt

Found in: Component config > ESP System Settings > CONFIG_ESP_INT_WDT

Also detect if interrupts on CPU 1 are disabled for too long.

Default value:
- Yes (enabled) if *CONFIG_ESP_INT_WDT* && *CONFIG_FREERTOS_UNICORE*

CONFIG_ESP_TASK_WDT_EN

Enable Task Watchdog Timer

Found in: Component config > ESP System Settings

The Task Watchdog Timer can be used to make sure individual tasks are still running. Enabling this option will enable the Task Watchdog Timer. It can be either initialized automatically at startup or initialized after startup (see Task Watchdog Timer API Reference)

Default value:
- Yes (enabled)

CONFIG_ESP_TASK_WDT_INIT

Initialize Task Watchdog Timer on startup

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN

Enabling this option will cause the Task Watchdog Timer to be initialized automatically at startup.

Default value:
- Yes (enabled)

CONFIG_ESP_TASK_WDT_PANIC

Invoke panic handler on Task Watchdog timeout

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

If this option is enabled, the Task Watchdog Timer will be configured to trigger the panic handler when it times out. This can also be configured at run time (see Task Watchdog Timer API Reference)

Default value:
- No (disabled)

CONFIG_ESP_TASK_WDT_TIMEOUT_S

Task Watchdog timeout period (seconds)

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

Timeout period configuration for the Task Watchdog Timer in seconds. This is also configurable at run time (see Task Watchdog Timer API Reference)

Range:
- from 1 to 60

Default value:
- 5

CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0

Watch CPU0 Idle Task

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

If this option is enabled, the Task Watchdog Timer will watch the CPU0 Idle Task. Having the Task Watchdog watch the Idle Task allows for detection of CPU starvation as the Idle Task not being called is usually a symptom of CPU starvation. Starvation of the Idle Task is detrimental as FreeRTOS household tasks depend on the Idle Task getting some runtime every now and then.

Default value:
- Yes (enabled)

CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1

Watch CPU1 Idle Task

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

If this option is enabled, the Task Watchdog Timer will wach the CPU1 Idle Task.

Default value:
- Yes (enabled) if *CONFIG_ESP_TASK_WDT_INIT* && *CONFIG_FREERTOS_UNICORE*

CONFIG_ESP_PANIC_HANDLER_IRAM

Place panic handler code in IRAM

Found in: Component config > ESP System Settings

If this option is disabled (default), the panic handler code is placed in flash not IRAM. This means that if ESP-IDF crashes while flash cache is disabled, the panic handler will automatically re-enable flash cache before running GDB Stub or Core Dump. This adds some minor risk, if the flash cache status is also corrupted during the crash.

If this option is enabled, the panic handler code (including required UART functions) is placed in IRAM. This may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash) or when flash cache is corrupted when an exception is triggered.

Default value:
- No (disabled)

CONFIG_ESP_DEBUG_STUBS_ENABLE

OpenOCD debug stubs

Found in: Component config > ESP System Settings

Debug stubs are used by OpenOCD to execute pre-compiled onboard code which does some useful debugging stuff, e.g. GCOV data dump.

Default value:

- "COMPILER_OPTIMIZATION_LEVEL_DEBUG" if *CONFIG_ESP32_TRAX* && ESP32S2_TRAX && ESP32S3_TRAX

CONFIG_ESP_DEBUG_OCDAWARE

Make exception and panic handlers JTAG/OCD aware

Found in: Component config > ESP System Settings

The FreeRTOS panic and unhandled exception handers can detect a JTAG OCD debugger and instead of panicking, have the debugger stop on the offending instruction.

Default value:

- Yes (enabled)

CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL

Interrupt level to use for Interrupt Watchdog and other system checks

Found in: Component config > ESP System Settings

Interrupt level to use for Interrupt Watchdog and other system checks.

Available options:

- Level 5 interrupt (ESP_SYSTEM_CHECK_INT_LEVEL_5)
  Using level 5 interrupt for Interrupt Watchdog and other system checks.
- Level 4 interrupt (ESP_SYSTEM_CHECK_INT_LEVEL_4)
  Using level 4 interrupt for Interrupt Watchdog and other system checks.

Brownout Detector Contains:

- *CONFIG_ESP_BROWNOUT_DET*

CONFIG_ESP_BROWNOUT_DET

Hardware brownout detect & reset

Found in: Component config > ESP System Settings > Brownout Detector

The ESP has a built-in brownout detector which can detect if the voltage is lower than a specific value. If this happens, it will reset the chip in order to prevent unintended behaviour.

Default value:

- Yes (enabled)

CONFIG_ESP_BROWNOUT_DET_LVL_SEL

Brownout voltage level

Found in: Component config > ESP System Settings > Brownout Detector > CONFIG_ESP_BROWNOUT_DET

The brownout detector will reset the chip when the supply voltage is approximately below this level. Note that there may be some variation of brownout voltage level between each ESP chip.

#The voltage levels here are estimates, more work needs to be done to figure out the exact voltages #of the brownout threshold levels.

Available options:

- 2.43V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_0)
- 2.48V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_1)
- 2.58V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_2)
- 2.62V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_3)
- 2.67V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_4)
- 2.70V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_5)
- 2.77V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_6)
- 2.80V +/- 0.05 (ESP_BROWNOUT_DET_LVL_SEL_7)

CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE

Permanently disable BASIC ROM Console

Found in: Component config > ESP System Settings

If set, the first time the app boots it will disable the BASIC ROM Console permanently (by burning an eFuse).

Otherwise, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

(Enabling secure boot also disables the BASIC ROM Console by default.)

Default value:

- No (disabled)

IPC (Inter-Processor Call)

Contains:

- *CONFIG_ESP_IPC_TASK_STACK_SIZE*
- *CONFIG_ESP_IPC_USES_CALLERS_PRIORITY*

CONFIG_ESP_IPC_TASK_STACK_SIZE

Inter-Processor Call (IPC) task stack size

Found in: Component config > IPC (Inter-Processor Call)

Configure the IPC tasks stack size. An IPC task runs on each core (in dual core mode), and allows for cross-core function calls. See IPC documentation for more details. The default IPC stack size should be enough for most common simple use cases. However, users can increase/decrease the stack size to their needs.

Range:

- from 512 to 65536

Default value:

- 1024

CONFIG_ESP_IPC_USES_CALLERS_PRIORITY

IPC runs at caller’ s priority

Found in: Component config > IPC (Inter-Processor Call)

If this option is not enabled then the IPC task will keep behavior same as prior to that of ESP-IDF v4.0, hence IPC task will run at (configMAX_PRIORITIES - 1) priority.

Default value:

- Yes (enabled) if *CONFIG_FREERTOS_UNICORE*

High resolution timer (esp_timer) Contains:

- *CONFIG_ESP_TIMER_PROFILING*
- *CONFIG_ESP_TIMER_TASK_STACK_SIZE*
- *CONFIG_ESP_TIMER_INTERRUPT_LEVEL*
- *CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD*

CONFIG_ESP_TIMER_PROFILING

Enable esp_timer profiling features

Found in: Component config > High resolution timer (esp_timer)

If enabled, esp_timer_dump will dump information such as number of times the timer was started, number of times the timer has triggered, and the total time it took for the callback to run. This option has some effect on timer performance and the amount of memory used for timer storage, and should only be used for debugging/testing purposes.

Default value:

- No (disabled)

CONFIG_ESP_TIMER_TASK_STACK_SIZE

High-resolution timer task stack size

Found in: Component config > High resolution timer (esp_timer)

Configure the stack size of "timer_task" task. This task is used to dispatch callbacks of timers created using ets_timer and esp_timer APIs. If you are seing stack overflow errors in timer task, increase this value.

Note that this is not the same as FreeRTOS timer task. To configure FreeRTOS timer task size, see "FreeRTOS timer task stack size" option in "FreeRTOS" menu.

Range:

- from 2048 to 65536

Default value:

- 3584

CONFIG_ESP_TIMER_INTERRUPT_LEVEL

Interrupt level

Found in: Component config > High resolution timer (esp_timer)

It sets the interrupt level for esp_timer ISR in range 1..3. A higher level (3) helps to decrease the ISR esp_timer latency.

Range:

- from 1 to 3
- from 1 to 1

Default value:

- 1

CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD

Support ISR dispatch method

Found in: Component config > High resolution timer (esp_timer)

Allows using ESP_TIMER_ISR dispatch method (ESP_TIMER_TASK dispatch method is also available). - ESP_TIMER_TASK - Timer callbacks are dispatched from a high-priority esp_timer task. - ESP_TIMER_ISR - Timer callbacks are dispatched directly from the timer interrupt handler. The ISR dispatch can be used, in some cases, when a callback is very simple or need a lower-latency.

Default value:

- No (disabled)

Wi-Fi Contains:

- *CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT*
- *CONFIG_ESP32_WIFI_ENABLE_WPA3_OWE_STA*
- *CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE*
- *CONFIG_ESP_WIFI_WPS_PASSPHRASE*
- *CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN*
- *CONFIG_ESP32_WIFI_CACHE_TX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM*
- *CONFIG_ESP_WIFI_RX_MGMT_BUF_NUM_DEF*
- *CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM*
- *CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM*
- *CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE*
- *CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE*
- *CONFIG_ESP_WIFI_MGMT_RX_BUFFER*
- *CONFIG_ESP32_WIFI_TX_BUFFER*
- *CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED*
- *CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED*
- *CONFIG_ESP32_WIFI_AMSDU_TX_ENABLED*
- *CONFIG_ESP32_WIFI_CSI_ENABLED*
- *CONFIG_ESP_WIFI_FTM_ENABLE*
- *CONFIG_ESP_WIFI_GCMP_SUPPORT*
- *CONFIG_ESP_WIFI_GMAC_SUPPORT*
- *CONFIG_ESP32_WIFI_IRAM_OPT*
- *CONFIG_ESP32_WIFI_MGMT_SBUF_NUM*
- *CONFIG_ESP32_WIFI_NVS_ENABLED*
- *CONFIG_ESP32_WIFI_RX_IRAM_OPT*
- *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*
- *CONFIG_ESP_WIFI_SLP_IRAM_OPT*
- *CONFIG_ESP_WIFI_SOFTAP_SUPPORT*
- *CONFIG_ESP32_WIFI_TASK_CORE_ID*

CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE

Software controls WiFi/Bluetooth coexistence

Found in: Component config > Wi-Fi

If enabled, WiFi & Bluetooth coexistence is controlled by software rather than hardware. Recommended for heavy traffic scenarios. Both coexistence configuration options are automatically managed, no user intervention is required. If only Bluetooth is used, it is recommended to disable this option to reduce binary file size.

Default value:

- Yes (enabled) if *CONFIG_BT_ENABLED*

CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM

Max number of WiFi static RX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi static RX buffers. Each buffer takes approximately 1.6KB of RAM. The static rx buffers are allocated when esp_wifi_init is called, they are not freed until esp_wifi_deinit is called.

WiFi hardware use these buffers to receive all 802.11 frames. A higher number may allow higher throughput but increases memory use. If ESP32_WIFI_AMPDU_RX_ENABLED is enabled, this value is recommended to set equal or bigger than ESP32_WIFI_RX_BA_WIN in order to achieve better throughput and compatibility with both stations and APs.

Range:

- from 2 to 25

Default value:

- 10 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*
- 16 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*

CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM

Max number of WiFi dynamic RX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi dynamic RX buffers, 0 means unlimited RX buffers will be allocated (provided sufficient free RAM). The size of each dynamic RX buffer depends on the size of the received data frame.

For each received data frame, the WiFi driver makes a copy to an RX buffer and then delivers it to the high layer TCP/IP stack. The dynamic RX buffer is freed after the higher layer has successfully received the data frame.

For some applications, WiFi data frames may be received faster than the application can process them. In these cases we may run out of memory if RX buffer number is unlimited (0).

If a dynamic RX buffer limit is set, it should be at least the number of static RX buffers.

Range:

- from 0 to 128 if *CONFIG_LWIP_WND_SCALE*
- from 0 to 1024 if *CONFIG_LWIP_WND_SCALE*

Default value:

- 32

CONFIG_ESP32_WIFI_TX_BUFFER

Type of WiFi TX buffers

Found in: Component config > Wi-Fi

Select type of WiFi TX buffers:

If "Static" is selected, WiFi TX buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. The size of each static TX buffer is fixed to about 1.6KB.

If "Dynamic" is selected, each WiFi TX buffer is allocated as needed when a data frame is delivered to the Wifi driver from the TCP/IP stack. The buffer is freed after the data frame has been sent by the WiFi driver. The size of each dynamic TX buffer depends on the length of each data frame sent by the TCP/IP layer.

If PSRAM is enabled, "Static" should be selected to guarantee enough WiFi TX buffers. If PSRAM is disabled, "Dynamic" should be selected to improve the utilization of RAM.

Available options:

- Static (ESP32_WIFI_STATIC_TX_BUFFER)
- Dynamic (ESP32_WIFI_DYNAMIC_TX_BUFFER)

CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM

Max number of WiFi static TX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi static TX buffers. Each buffer takes approximately 1.6KB of RAM. The static RX buffers are allocated when esp_wifi_init() is called, they are not released until esp_wifi_deinit() is called.

For each transmitted data frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

Range:
- from 1 to 64 if ESP32_WIFI_STATIC_TX_BUFFER

Default value:
- 16 if ESP32_WIFI_STATIC_TX_BUFFER

CONFIG_ESP32_WIFI_CACHE_TX_BUFFER_NUM

Max number of WiFi cache TX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi cache TX buffer number.

For each TX packet from uplayer, such as LWIP etc, WiFi driver needs to allocate a static TX buffer and makes a copy of uplayer packet. If WiFi driver fails to allocate the static TX buffer, it caches the uplayer packets to a dedicated buffer queue, this option is used to configure the size of the cached TX queue.

Range:
- from 16 to 128 if *CONFIG_SPIRAM*

Default value:
- 32 if *CONFIG_SPIRAM*

CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM

Max number of WiFi dynamic TX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi dynamic TX buffers. The size of each dynamic TX buffer is not fixed, it depends on the size of each transmitted data frame.

For each transmitted frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications, especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

Range:
- from 1 to 128

Default value:
- 32

CONFIG_ESP_WIFI_MGMT_RX_BUFFER

Type of WiFi RX MGMT buffers

Found in: Component config > Wi-Fi

Select type of WiFi RX MGMT buffers:

If "Static" is selected, WiFi RX MGMT buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. The size of each static RX MGMT buffer is fixed to about 500 Bytes.

If "Dynamic" is selected, each WiFi RX MGMT buffer is allocated as needed when a MGMT data frame is received. The MGMT buffer is freed after the MGMT data frame has been processed by the WiFi driver.

Available options:

- Static (ESP_WIFI_STATIC_RX_MGMT_BUFFER)
- Dynamic (ESP_WIFI_DYNAMIC_RX_MGMT_BUFFER)

CONFIG_ESP_WIFI_RX_MGMT_BUF_NUM_DEF

Max number of WiFi RX MGMT buffers

Found in: Component config > Wi-Fi

Set the number of WiFi RX_MGMT buffers.

For Management buffers, the number of dynamic and static management buffers is the same. In order to prevent memory fragmentation, the management buffer type should be set to static first.

Range:
- from 1 to 10

Default value:
- 5

CONFIG_ESP32_WIFI_CSI_ENABLED

WiFi CSI(Channel State Information)

Found in: Component config > Wi-Fi

Select this option to enable CSI(Channel State Information) feature. CSI takes about CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM KB of RAM. If CSI is not used, it is better to disable this feature in order to save memory.

Default value:
- No (disabled)

CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED

WiFi AMPDU TX

Found in: Component config > Wi-Fi

Select this option to enable AMPDU TX feature

Default value:
- Yes (enabled)

CONFIG_ESP32_WIFI_TX_BA_WIN

WiFi AMPDU TX BA window size

Found in: Component config > Wi-Fi > CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED

Set the size of WiFi Block Ack TX window. Generally a bigger value means higher throughput but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP TX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12.

Range:
- from 2 to 32

Default value:
- 6

CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED

WiFi AMPDU RX

Found in: Component config > Wi-Fi

Select this option to enable AMPDU RX feature

Default value:

- Yes (enabled)

CONFIG_ESP32_WIFI_RX_BA_WIN

WiFi AMPDU RX BA window size

Found in: Component config > Wi-Fi > CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED

Set the size of WiFi Block Ack RX window. Generally a bigger value means higher throughput and better compatibility but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP RX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12. If PSRAM is used and WiFi memory is prefered to allocat in PSRAM first, the default and minimum value should be 16 to achieve better throughput and compatibility with both stations and APs.

Range:

- from 2 to 32

Default value:

- 6 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP* && *CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED*
- 16 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP* && *CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED*

CONFIG_ESP32_WIFI_AMSDU_TX_ENABLED

WiFi AMSDU TX

Found in: Component config > Wi-Fi

Select this option to enable AMSDU TX feature

Default value:

- No (disabled) if *CONFIG_SPIRAM*

CONFIG_ESP32_WIFI_NVS_ENABLED

WiFi NVS flash

Found in: Component config > Wi-Fi

Select this option to enable WiFi NVS flash

Default value:

- Yes (enabled)

CONFIG_ESP32_WIFI_TASK_CORE_ID

WiFi Task Core ID

Found in: Component config > Wi-Fi

Pinned WiFi task to core 0 or core 1.

Available options:

- Core 0 (ESP32_WIFI_TASK_PINNED_TO_CORE_0)
- Core 1 (ESP32_WIFI_TASK_PINNED_TO_CORE_1)

CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN

Max length of WiFi SoftAP Beacon

Found in: Component config > Wi-Fi

ESP-MESH utilizes beacon frames to detect and resolve root node conflicts (see documentation). However the default length of a beacon frame can simultaneously hold only five root node identifier structures, meaning that a root node conflict of up to five nodes can be detected at one time. In the occurence of more root nodes conflict involving more than five root nodes, the conflict resolution process will detect five of the root nodes, resolve the conflict, and re-detect more root nodes. This process will repeat until all root node conflicts are resolved. However this process can generally take a very long time.

To counter this situation, the beacon frame length can be increased such that more root nodes can be detected simultaneously. Each additional root node will require 36 bytes and should be added ontop of the default beacon frame length of 752 bytes. For example, if you want to detect 10 root nodes simultaneously, you need to set the beacon frame length as 932 (752+36*5).

Setting a longer beacon length also assists with debugging as the conflicting root nodes can be identified more quickly.

Range:
- from 752 to 1256

Default value:
- 752

CONFIG_ESP32_WIFI_MGMT_SBUF_NUM

WiFi mgmt short buffer number

Found in: Component config > Wi-Fi

Set the number of WiFi management short buffer.

Range:
- from 6 to 32

Default value:
- 32

CONFIG_ESP32_WIFI_IRAM_OPT

WiFi IRAM speed optimization

Found in: Component config > Wi-Fi

Select this option to place frequently called Wi-Fi library functions in IRAM. When this option is disabled, more than 10Kbytes of IRAM memory will be saved but Wi-Fi throughput will be reduced.

Default value:
- No (disabled) if *CONFIG_BT_ENABLED* && *CONFIG_SPIRAM*
- Yes (enabled)

CONFIG_ESP32_WIFI_RX_IRAM_OPT

WiFi RX IRAM speed optimization

Found in: Component config > Wi-Fi

Select this option to place frequently called Wi-Fi library RX functions in IRAM. When this option is disabled, more than 17Kbytes of IRAM memory will be saved but Wi-Fi performance will be reduced.

Default value:
- No (disabled) if *CONFIG_BT_ENABLED* && *CONFIG_SPIRAM*
- Yes (enabled)

CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE

Enable WPA3-Personal

Found in: Component config > Wi-Fi

Select this option to allow the device to establish a WPA3-Personal connection with eligible AP's. PMF (Protected Management Frames) is a prerequisite feature for a WPA3 connection, it needs to be explicitly configured before attempting connection. Please refer to the Wi-Fi Driver API Guide for details.

Default value:

- Yes (enabled)

CONFIG_ESP32_WIFI_ENABLE_WPA3_OWE_STA

Enable OWE STA

Found in: Component config > Wi-Fi

Select this option to allow the device to establish OWE connection with eligible AP's. PMF (Protected Management Frames) is a prerequisite feature for a WPA3 connection, it needs to be explicitly configured before attempting connection. Please refer to the Wi-Fi Driver API Guide for details.

Default value:

- Yes (enabled)

CONFIG_ESP_WIFI_SLP_IRAM_OPT

WiFi SLP IRAM speed optimization

Found in: Component config > Wi-Fi

Select this option to place called Wi-Fi library TBTT process and receive beacon functions in IRAM. Some functions can be put in IRAM either by ESP32_WIFI_IRAM_OPT and ESP32_WIFI_RX_IRAM_OPT, or this one. If already enabled ESP32_WIFI_IRAM_OPT, the other 7.3KB IRAM memory would be taken by this option. If already enabled ESP32_WIFI_RX_IRAM_OPT, the other 1.3KB IRAM memory would be taken by this option. If neither of them are enabled, the other 7.4KB IRAM memory would be taken by this option. Wi-Fi power-save mode average current would be reduced if this option is enabled.

CONFIG_ESP_WIFI_SLP_DEFAULT_MIN_ACTIVE_TIME

Minimum active time

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_IRAM_OPT

The minimum timeout for waiting to receive data, unit: milliseconds.

Range:

- from 8 to 60 if *CONFIG_ESP_WIFI_SLP_IRAM_OPT*

Default value:

- 50 if *CONFIG_ESP_WIFI_SLP_IRAM_OPT*

CONFIG_ESP_WIFI_SLP_DEFAULT_MAX_ACTIVE_TIME

Maximum keep alive time

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_IRAM_OPT

The maximum time that wifi keep alive, unit: seconds.

Range:

- from 10 to 60 if *CONFIG_ESP_WIFI_SLP_IRAM_OPT*

Default value:

- 10 if *CONFIG_ESP_WIFI_SLP_IRAM_OPT*

CONFIG_ESP_WIFI_FTM_ENABLE

WiFi FTM

Found in: Component config > Wi-Fi

Enable feature Fine Timing Measurement for calculating WiFi Round-Trip-Time (RTT).

CONFIG_ESP_WIFI_FTM_INITIATOR_SUPPORT

FTM Initiator support

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_FTM_ENABLE

Default value:

- Yes (enabled) if *CONFIG_ESP_WIFI_FTM_ENABLE*

CONFIG_ESP_WIFI_FTM_RESPONDER_SUPPORT

FTM Responder support

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_FTM_ENABLE

Default value:

- Yes (enabled) if *CONFIG_ESP_WIFI_FTM_ENABLE*

CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE

Power Management for station at disconnected

Found in: Component config > Wi-Fi

Select this option to enable power_management for station when disconnected. Chip will do modem-sleep when rf module is not in use any more.

Default value:

- Yes (enabled)

CONFIG_ESP_WIFI_GCMP_SUPPORT

WiFi GCMP Support(GCMP128 and GCMP256)

Found in: Component config > Wi-Fi

Select this option to enable GCMP support. GCMP support is compulsory for WiFi Suite-B support.

CONFIG_ESP_WIFI_GMAC_SUPPORT

WiFi GMAC Support(GMAC128 and GMAC256)

Found in: Component config > Wi-Fi

Select this option to enable GMAC support. GMAC support is compulsory for WiFi 192 bit certification.

Default value:

- No (disabled)

CONFIG_ESP_WIFI_SOFTAP_SUPPORT

WiFi SoftAP Support

Found in: Component config > Wi-Fi

WiFi module can be compiled without SoftAP to save code size.

Default value:

- Yes (enabled)

CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

Wifi sleep optimize when beacon lost

Found in: Component config > Wi-Fi

Enable wifi sleep optimization when beacon loss occurs and immediately enter sleep mode when the WiFi module detects beacon loss.

CONFIG_ESP_WIFI_SLP_BEACON_LOST_TIMEOUT

Beacon loss timeout

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

Timeout time for close rf phy when beacon loss occurs, Unit: 1024 microsecond.

Range:

- from 5 to 100 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Default value:

- 10 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

CONFIG_ESP_WIFI_SLP_BEACON_LOST_THRESHOLD

Maximum number of consecutive lost beacons allowed

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

Maximum number of consecutive lost beacons allowed, WiFi keeps Rx state when the number of consecutive beacons lost is greater than the given threshold.

Range:

- from 0 to 8 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Default value:

- 3 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

CONFIG_ESP_WIFI_SLP_PHY_ON_DELTA_EARLY_TIME

Delta early time for RF PHY on

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

Delta early time for rf phy on, When the beacon is lost, the next rf phy on will be earlier the time specified by the configuration item, Unit: 32 microsecond.

Range:

- from 0 to 100 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Default value:

- 2 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

CONFIG_ESP_WIFI_SLP_PHY_OFF_DELTA_TIMEOUT_TIME

Delta timeout time for RF PHY off

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

Delta timeout time for rf phy off, When the beacon is lost, the next rf phy off will be delayed for the time specified by the configuration item. Unit: 1024 microsecond.

Range:
- from 0 to 8 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Default value:
- 2 if *CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM

Maximum espnow encrypt peers number

Found in: Component config > Wi-Fi

Maximum number of encrypted peers supported by espnow. The number of hardware keys for encryption is fixed. And the espnow and SoftAP share the same hardware keys. So this configuration will affect the maximum connection number of SoftAP. Maximum espnow encrypted peers number + maximum number of connections of SoftAP = Max hardware keys number.

When using ESP mesh, this value should be set to a maximum of 6.

Range:
- from 0 to 17

Default value:
- 7

CONFIG_ESP_WIFI_WPS_PASSPHRASE

Get WPA2 passphrase in WPS config

Found in: Component config > Wi-Fi

Select this option to get passphrase during WPS configuration. This option fakes the virtual display capabilites to get the configuration in passphrase mode. Not recommanded to be used since WPS credentials should not be shared to other devices, making it in readable format increases that risk, also passphrase requires pbkdf2 to convert in psk.

Default value:
- No (disabled)

CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT

Enable enterprise option

Found in: Component config > Wi-Fi

Select this to enable/disable enterprise connection support.

disabling this will reduce binary size. disabling this will disable the use of any esp_wifi_sta_wpa2_ent_* (as APIs will be meaningless)

Default value:
- Yes (enabled)

CONFIG_ESP_WIFI_ENT_FREE_DYNAMIC_BUFFER

Free dynamic buffers during WiFi enterprise connection

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT

Select this configuration to free dynamic buffers during WiFi enterprise connection. This will enable chip to reduce heap consumption during WiFi enterprise connection.

Default value:

- No (disabled)

Core dump Contains:

- *CONFIG_ESP_COREDUMP_CHECK_BOOT*
- *CONFIG_ESP_COREDUMP_DATA_FORMAT*
- *CONFIG_ESP_COREDUMP_CHECKSUM*
- *CONFIG_ESP_COREDUMP_TO_FLASH_OR_UART*
- *CONFIG_ESP_COREDUMP_UART_DELAY*
- *CONFIG_ESP_COREDUMP_LOGS*
- *CONFIG_ESP_COREDUMP_DECODE*
- *CONFIG_ESP_COREDUMP_MAX_TASKS_NUM*
- *CONFIG_ESP_COREDUMP_STACK_SIZE*

CONFIG_ESP_COREDUMP_TO_FLASH_OR_UART

Data destination

Found in: Component config > Core dump

Select place to store core dump: flash, uart or none (to disable core dumps generation).

Core dumps to Flash are not available if PSRAM is used for task stacks.

If core dump is configured to be stored in flash and custom partition table is used add corresponding entry to your CSV. For examples, please see predefined partition table CSV descriptions in the components/partition_table directory.

Available options:

- Flash (ESP_COREDUMP_ENABLE_TO_FLASH)
- UART (ESP_COREDUMP_ENABLE_TO_UART)
- None (ESP_COREDUMP_ENABLE_TO_NONE)

CONFIG_ESP_COREDUMP_DATA_FORMAT

Core dump data format

Found in: Component config > Core dump

Select the data format for core dump.

Available options:

- Binary format (ESP_COREDUMP_DATA_FORMAT_BIN)
- ELF format (ESP_COREDUMP_DATA_FORMAT_ELF)

CONFIG_ESP_COREDUMP_CHECKSUM

Core dump data integrity check

Found in: Component config > Core dump

Select the integrity check for the core dump.

Available options:

- Use CRC32 for integrity verification (ESP_COREDUMP_CHECKSUM_CRC32)

- Use SHA256 for integrity verification (ESP_COREDUMP_CHECKSUM_SHA256)

CONFIG_ESP_COREDUMP_CHECK_BOOT

Check core dump data integrity on boot

Found in: Component config > Core dump

When enabled, if any data are found on the flash core dump partition, they will be checked by calculating their checksum.

Default value:

- Yes (enabled) if ESP_COREDUMP_ENABLE_TO_FLASH

CONFIG_ESP_COREDUMP_LOGS

Enable coredump logs for debugging

Found in: Component config > Core dump

Enable/disable coredump logs. Logs strings from espcoredump component are placed in DRAM. Disabling these helps to save ~5KB of internal memory.

CONFIG_ESP_COREDUMP_MAX_TASKS_NUM

Maximum number of tasks

Found in: Component config > Core dump

Maximum number of tasks snapshots in core dump.

CONFIG_ESP_COREDUMP_UART_DELAY

Delay before print to UART

Found in: Component config > Core dump

Config delay (in ms) before printing core dump to UART. Delay can be interrupted by pressing Enter key.

Default value:

- 0 if ESP_COREDUMP_ENABLE_TO_UART

CONFIG_ESP_COREDUMP_STACK_SIZE

Reserved stack size

Found in: Component config > Core dump

Size of the memory to be reserved for core dump stack. If 0 core dump process will run on the stack of crashed task/ISR, otherwise special stack will be allocated. To ensure that core dump itself will not overflow task/ISR stack set this to the value above 800. NOTE: It eats DRAM.

CONFIG_ESP_COREDUMP_DECODE

Handling of UART core dumps in IDF Monitor

Found in: Component config > Core dump

Available options:

- Decode and show summary (info_corefile) (ESP_COREDUMP_DECODE_INFO)
- Don’t decode (ESP_COREDUMP_DECODE_DISABLE)

FAT Filesystem support Contains:

- *CONFIG_FATFS_API_ENCODING*
- *CONFIG_FATFS_USE_FASTSEEK*
- *CONFIG_FATFS_CHOOSE_TYPE*
- *CONFIG_FATFS_LONG_FILENAMES*
- *CONFIG_FATFS_MAX_LFN*
- *CONFIG_FATFS_FS_LOCK*
- *CONFIG_FATFS_VOLUME_COUNT*
- *CONFIG_FATFS_CHOOSE_CODEPAGE*
- *CONFIG_FATFS_ALLOC_PREFER_EXTRAM*
- *CONFIG_FATFS_SECTOR_SIZE*
- *CONFIG_FATFS_SECTORS_PER_CLUSTER*
- *CONFIG_FATFS_TIMEOUT_MS*
- *CONFIG_FATFS_PER_FILE_CACHE*

CONFIG_FATFS_VOLUME_COUNT

Number of volumes

Found in: Component config > FAT Filesystem support

Number of volumes (logical drives) to use.

Range:
- from 1 to 10

Default value:
- 2

CONFIG_FATFS_SECTOR_SIZE

Sector size

Found in: Component config > FAT Filesystem support

Specify the size of the sector in bytes for FATFS partition generator.

Available options:
- 512 (FATFS_SECTOR_512)
- 1024 (FATFS_SECTOR_1024)
- 2048 (FATFS_SECTOR_2048)
- 4096 (FATFS_SECTOR_4096)

CONFIG_FATFS_SECTORS_PER_CLUSTER

Sectors per cluster

Found in: Component config > FAT Filesystem support

This value specifies how many sectors there are in one cluster.

Available options:
- 1 (FATFS_SECTORS_PER_CLUSTER_1)
- 2 (FATFS_SECTORS_PER_CLUSTER_2)
- 4 (FATFS_SECTORS_PER_CLUSTER_4)
- 8 (FATFS_SECTORS_PER_CLUSTER_8)
- 16 (FATFS_SECTORS_PER_CLUSTER_16)
- 32 (FATFS_SECTORS_PER_CLUSTER_32)
- 64 (FATFS_SECTORS_PER_CLUSTER_64)
- 128 (FATFS_SECTORS_PER_CLUSTER_128)

CONFIG_FATFS_CHOOSE_CODEPAGE

OEM Code Page

Found in: Component config > FAT Filesystem support

OEM code page used for file name encodings.

If "Dynamic" is selected, code page can be chosen at runtime using f_setcp function. Note that choosing this option will increase application size by ~480kB.

Available options:

- Dynamic (all code pages supported) (FATFS_CODEPAGE_DYNAMIC)
- US (CP437) (FATFS_CODEPAGE_437)
- Arabic (CP720) (FATFS_CODEPAGE_720)
- Greek (CP737) (FATFS_CODEPAGE_737)
- KBL (CP771) (FATFS_CODEPAGE_771)
- Baltic (CP775) (FATFS_CODEPAGE_775)
- Latin 1 (CP850) (FATFS_CODEPAGE_850)
- Latin 2 (CP852) (FATFS_CODEPAGE_852)
- Cyrillic (CP855) (FATFS_CODEPAGE_855)
- Turkish (CP857) (FATFS_CODEPAGE_857)
- Portugese (CP860) (FATFS_CODEPAGE_860)
- Icelandic (CP861) (FATFS_CODEPAGE_861)
- Hebrew (CP862) (FATFS_CODEPAGE_862)
- Canadian French (CP863) (FATFS_CODEPAGE_863)
- Arabic (CP864) (FATFS_CODEPAGE_864)
- Nordic (CP865) (FATFS_CODEPAGE_865)
- Russian (CP866) (FATFS_CODEPAGE_866)
- Greek 2 (CP869) (FATFS_CODEPAGE_869)
- Japanese (DBCS) (CP932) (FATFS_CODEPAGE_932)
- Simplified Chinese (DBCS) (CP936) (FATFS_CODEPAGE_936)
- Korean (DBCS) (CP949) (FATFS_CODEPAGE_949)
- Traditional Chinese (DBCS) (CP950) (FATFS_CODEPAGE_950)

CONFIG_FATFS_CHOOSE_TYPE

FAT type

Found in: Component config > FAT Filesystem support

If user specifies automatic detection of the FAT type, the FATFS generator will determine the type by the size.

Available options:

- Select a suitable FATFS type automatically. (FATFS_AUTO_TYPE)
- FAT12 (FATFS_FAT12)
- FAT16 (FATFS_FAT16)

CONFIG_FATFS_LONG_FILENAMES

Long filename support

Found in: Component config > FAT Filesystem support

Support long filenames in FAT. Long filename data increases memory usage. FATFS can be configured to store the buffer for long filename data in stack or heap (Currently not supported by FATFS partition generator).

Available options:

- No long filenames (FATFS_LFN_NONE)
- Long filename buffer in heap (FATFS_LFN_HEAP)
- Long filename buffer on stack (FATFS_LFN_STACK)

CONFIG_FATFS_MAX_LFN

Max long filename length

Found in: Component config > FAT Filesystem support

Maximum long filename length. Can be reduced to save RAM.

Range:
- from 12 to 255

Default value:
- 255

CONFIG_FATFS_API_ENCODING

API character encoding

Found in: Component config > FAT Filesystem support

Choose encoding for character and string arguments/returns when using FATFS APIs. The encoding of arguments will usually depend on text editor settings.

Available options:
- API uses ANSI/OEM encoding (FATFS_API_ENCODING_ANSI_OEM)
- API uses UTF-8 encoding (FATFS_API_ENCODING_UTF_8)

CONFIG_FATFS_FS_LOCK

Number of simultaneously open files protected by lock function

Found in: Component config > FAT Filesystem support

This option sets the FATFS configuration value _FS_LOCK. The option _FS_LOCK switches file lock function to control duplicated file open and illegal operation to open objects.

* 0: Disable file lock function. To avoid volume corruption, application should avoid illegal open, remove and rename to the open objects.

* >0: Enable file lock function. The value defines how many files/sub-directories can be opened simultaneously under file lock control.

Note that the file lock control is independent of re-entrancy.

Range:
- from 0 to 65535

Default value:
- 0

CONFIG_FATFS_TIMEOUT_MS

Timeout for acquiring a file lock, ms

Found in: Component config > FAT Filesystem support

This option sets FATFS configuration value _FS_TIMEOUT, scaled to milliseconds. Sets the number of milliseconds FATFS will wait to acquire a mutex when operating on an open file. For example, if one task is performing a lenghty operation, another task will wait for the first task to release the lock, and time out after amount of time set by this option.

Default value:
- 10000

CONFIG_FATFS_PER_FILE_CACHE

Use separate cache for each file

Found in: Component config > FAT Filesystem support

This option affects FATFS configuration value _FS_TINY.

If this option is set, _FS_TINY is 0, and each open file has its own cache, size of the cache is equal to the _MAX_SS variable (512 or 4096 bytes). This option uses more RAM if more than 1 file is open, but needs less reads and writes to the storage for some operations.

If this option is not set, _FS_TINY is 1, and single cache is used for all open files, size is also equal to _MAX_SS variable. This reduces the amount of heap used when multiple files are open, but increases the number of read and write operations which FATFS needs to make.

Default value:

- Yes (enabled)

CONFIG_FATFS_ALLOC_PREFER_EXTRAM

Perfer external RAM when allocating FATFS buffers

Found in: Component config > FAT Filesystem support

When the option is enabled, internal buffers used by FATFS will be allocated from external RAM. If the allocation from external RAM fails, the buffer will be allocated from the internal RAM. Disable this option if optimizing for performance. Enable this option if optimizing for internal memory size.

Default value:

- Yes (enabled) if SPIRAM_USE_CAPS_ALLOC || SPIRAM_USE_MALLOC

CONFIG_FATFS_USE_FASTSEEK

Enable fast seek algorithm when using lseek function through VFS FAT

Found in: Component config > FAT Filesystem support

The fast seek feature enables fast backward/long seek operations without FAT access by using an in-memory CLMT (cluster link map table). Please note, fast-seek is only allowed for read-mode files, if a file is opened in write-mode, the seek mechanism will automatically fallback to the default implementation.

Default value:

- No (disabled)

CONFIG_FATFS_FAST_SEEK_BUFFER_SIZE

Fast seek CLMT buffer size

Found in: Component config > FAT Filesystem support > CONFIG_FATFS_USE_FASTSEEK

If fast seek algorithm is enabled, this defines the size of CLMT buffer used by this algorithm in 32-bit word units. This value should be chosen based on prior knowledge of maximum elements of each file entry would store.

Default value:

- 64 if *CONFIG_FATFS_USE_FASTSEEK*

FreeRTOS Contains:

- *Kernel*
- *Port*

Kernel Contains:

- *CONFIG_FREERTOS_CHECK_STACKOVERFLOW*
- *CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY*
- *CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS*
- *CONFIG_FREERTOS_MAX_TASK_NAME_LEN*
- *CONFIG_FREERTOS_IDLE_TASK_STACKSIZE*
- *CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS*
- *CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE*
- *CONFIG_FREERTOS_HZ*
- *CONFIG_FREERTOS_TIMER_QUEUE_LENGTH*
- *CONFIG_FREERTOS_TIMER_TASK_PRIORITY*
- *CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH*
- *CONFIG_FREERTOS_USE_IDLE_HOOK*
- *CONFIG_FREERTOS_OPTIMIZED_SCHEDULER*
- *CONFIG_FREERTOS_USE_TICK_HOOK*
- *CONFIG_FREERTOS_USE_TICKLESS_IDLE*
- *CONFIG_FREERTOS_USE_TRACE_FACILITY*
- *CONFIG_FREERTOS_UNICORE*
- *CONFIG_FREERTOS_SMP*
- *CONFIG_FREERTOS_USE_MINIMAL_IDLE_HOOK*

CONFIG_FREERTOS_SMP

Run the Amazon SMP FreeRTOS kernel instead (FEATURE UNDER DEVELOPMENT)

Found in: Component config > FreeRTOS > Kernel

Amazon has released an SMP version of the FreeRTOS Kernel which can be found via the following link: https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/smp

IDF has added an experimental port of this SMP kernel located in components/freertos/FreeRTOS-Kernel-SMP. Enabling this option will cause IDF to use the Amazon SMP kernel. Note that THIS FEATURE IS UNDER ACTIVE DEVELOPMENT, users use this at their own risk.

Leaving this option disabled will mean the IDF FreeRTOS kernel is used instead, which is located in: components/freertos/FreeRTOS-Kernel. Both kernel versions are SMP capable, but differ in their implementation and features.

Default value:

- No (disabled)

CONFIG_FREERTOS_UNICORE

Run FreeRTOS only on first core

Found in: Component config > FreeRTOS > Kernel

This version of FreeRTOS normally takes control of all cores of the CPU. Select this if you only want to start it on the first core. This is needed when e.g. another process needs complete control over the second core.

CONFIG_FREERTOS_HZ

configTICK_RATE_HZ

Found in: Component config > FreeRTOS > Kernel

Sets the FreeRTOS tick interrupt frequency in Hz (see configTICK_RATE_HZ documentation for more details).

Range:

- from 1 to 1000

Default value:

- 100

CONFIG_FREERTOS_OPTIMIZED_SCHEDULER

configUSE_PORT_OPTIMISED_TASK_SELECTION

Found in: Component config > FreeRTOS > Kernel

Enables port specific task selection method. This option can will speed up the search of ready tasks when scheduling (see configUSE_PORT_OPTIMISED_TASK_SELECTION documentation for more details).

Default value:

- Yes (enabled) if *CONFIG_FREERTOS_UNICORE*

CONFIG_FREERTOS_CHECK_STACKOVERFLOW

configCHECK_FOR_STACK_OVERFLOW

Found in: Component config > FreeRTOS > Kernel

Enables FreeRTOS to check for stack overflows (see configCHECK_FOR_STACK_OVERFLOW documentation for more details).

Note: If users do not provide their own `vApplicationStackOverflowHook()` function, a default function will be provided by ESP-IDF.

Available options:

- No checking (FREERTOS_CHECK_STACKOVERFLOW_NONE)
  Do not check for stack overflows (configCHECK_FOR_STACK_OVERFLOW = 0)
- Check by stack pointer value (Method 1) (FREERTOS_CHECK_STACKOVERFLOW_PTRVAL)
  Check for stack overflows on each context switch by checking if the stack pointer is in a valid range. Quick but does not detect stack overflows that happened between context switches (configCHECK_FOR_STACK_OVERFLOW = 1)
- Check using canary bytes (Method 2) (FREERTOS_CHECK_STACKOVERFLOW_CANARY)
  Places some magic bytes at the end of the stack area and on each context switch, check if these bytes are still intact. More thorough than just checking the pointer, but also slightly slower. (configCHECK_FOR_STACK_OVERFLOW = 2)

CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS

configNUM_THREAD_LOCAL_STORAGE_POINTERS

Found in: Component config > FreeRTOS > Kernel

Set the number of thread local storage pointers in each task (see configNUM_THREAD_LOCAL_STORAGE_POINTERS documentation for more details).

Note: In ESP-IDF, this value must be at least 1. Index 0 is reserved for use by the pthreads API thread-local-storage. Other indexes can be used for any desired purpose.

Range:

- from 1 to 256

Default value:

- 1

CONFIG_FREERTOS_IDLE_TASK_STACKSIZE

configMINIMAL_STACK_SIZE (Idle task stack size)

Found in: Component config > FreeRTOS > Kernel

Sets the idle task stack size in bytes (see configMINIMAL_STACK_SIZE documentation for more details).

Note:

- ESP-IDF specifies stack sizes in bytes instead of words.
- The default size is enough for most use cases.
- The stack size may need to be increased above the default if the app installs idle or thread local storage cleanup hooks that use a lot of stack memory.
- Conversely, the stack size can be reduced to the minimum if non of the idle features are used.

Range:
- from 768 to 32768

Default value:
- 1536

CONFIG_FREERTOS_USE_IDLE_HOOK

configUSE_IDLE_HOOK

Found in: Component config > FreeRTOS > Kernel

Enables the idle task application hook (see configUSE_IDLE_HOOK documentation for more details).

Note:

- The application must provide the hook function `void vApplicationIdleHook( void );`
- `vApplicationIdleHook()` is called from FreeRTOS idle task(s)
- The FreeRTOS idle hook is NOT the same as the ESP-IDF Idle Hook, but both can be enabled simultaneously.

Default value:
- No (disabled)

CONFIG_FREERTOS_USE_MINIMAL_IDLE_HOOK

Use FreeRTOS minimal idle hook

Found in: Component config > FreeRTOS > Kernel

Enables the minimal idle task application hook (see configUSE_IDLE_HOOK documentation for more details).

Note:

- The application must provide the hook function `void vApplicationMinimalIdleHook( void );`
- `vApplicationMinimalIdleHook()` is called from FreeRTOS minimal idle task(s)

Default value:
- No (disabled) if *CONFIG_FREERTOS_SMP*

CONFIG_FREERTOS_USE_TICK_HOOK

configUSE_TICK_HOOK

Found in: Component config > FreeRTOS > Kernel

Enables the tick hook (see configUSE_TICK_HOOK documentation for more details).

Note:

- The application must provide the hook function `void vApplicationTickHook( void );`

- `vApplicationTickHook()` is called from FreeRTOS' s tick handling function `xTaskIncrementTick()`
- The FreeRTOS tick hook is NOT the same as the ESP-IDF Tick Interrupt Hook, but both can be enabled simultaneously.

Default value:
- No (disabled)

CONFIG_FREERTOS_MAX_TASK_NAME_LEN

configMAX_TASK_NAME_LEN

Found in: Component config > FreeRTOS > Kernel

Sets the maximum number of characters for task names (see configMAX_TASK_NAME_LEN documentation for more details).

Note: For most uses, the default of 16 characters is sufficient.

Range:
- from 1 to 256

Default value:
- 16

CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY

configENABLE_BACKWARD_COMPATIBILITY

Found in: Component config > FreeRTOS > Kernel

Enable backward compatibility with APIs prior to FreeRTOS v8.0.0. (see configENABLE_BACKWARD_COMPATIBILITY documentation for more details).

Default value:
- No (disabled)

CONFIG_FREERTOS_TIMER_TASK_PRIORITY

configTIMER_TASK_PRIORITY

Found in: Component config > FreeRTOS > Kernel

Sets the timer task' s priority (see configTIMER_TASK_PRIORITY documentation for more details).

Range:
- from 1 to 25

Default value:
- 1

CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH

configTIMER_TASK_STACK_DEPTH

Found in: Component config > FreeRTOS > Kernel

Set the timer task' s stack size (see configTIMER_TASK_STACK_DEPTH documentation for more details).

Range:
- from 1536 to 32768

Default value:
- 2048

CONFIG_FREERTOS_TIMER_QUEUE_LENGTH

configTIMER_QUEUE_LENGTH

Found in: Component config > FreeRTOS > Kernel

Set the timer task ' s command queue length (see configTIMER_QUEUE_LENGTH documentation for more details).

Range:
- from 5 to 20

Default value:
- 10

CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE

configQUEUE_REGISTRY_SIZE

Found in: Component config > FreeRTOS > Kernel

Set the size of the queue registry (see configQUEUE_REGISTRY_SIZE documentation for more details).

Note: A value of 0 will disable queue registry functionality

Range:
- from 0 to 20

Default value:
- 0

CONFIG_FREERTOS_USE_TRACE_FACILITY

configUSE_TRACE_FACILITY

Found in: Component config > FreeRTOS > Kernel

Enables additional structure members and functions to assist with execution visualization and tracing (see configUSE_TRACE_FACILITY documentation for more details).

Default value:
- No (disabled)

CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS

configUSE_STATS_FORMATTING_FUNCTIONS

Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_USE_TRACE_FACILITY

Set configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS to 1 to include the `vTaskList()` and `vTaskGetRunTimeStats()` functions in the build (see configUSE_STATS_FORMATTING_FUNCTIONS documentation for more details).

Default value:
- No (disabled) if *CONFIG_FREERTOS_USE_TRACE_FACILITY*

CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID

Enable display of xCoreID in vTaskList

Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_USE_TRACE_FACILITY > CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS

If enabled, this will include an extra column when vTaskList is called to display the CoreID the task is pinned to (0,1) or -1 if not pinned.

Default value:

- No (disabled) if *CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS*

CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS

configGENERATE_RUN_TIME_STATS

Found in: Component config > FreeRTOS > Kernel

Enables collection of run time statistics for each task (see configGENERATE_RUN_TIME_STATS documentation for more details).

Note: The clock used for run time statistics can be configured in FREERTOS_RUN_TIME_STATS_CLK.

Default value:

- No (disabled)

CONFIG_FREERTOS_USE_TICKLESS_IDLE

configUSE_TICKLESS_IDLE

Found in: Component config > FreeRTOS > Kernel

If power management support is enabled, FreeRTOS will be able to put the system into light sleep mode when no tasks need to run for a number of ticks. This number can be set using FREERTOS_IDLE_TIME_BEFORE_SLEEP option. This feature is also known as "automatic light sleep" .

Note that timers created using esp_timer APIs may prevent the system from entering sleep mode, even when no tasks need to run. To skip unnecessary wake-up initialize a timer with the "skip_unhandled_events" option as true.

If disabled, automatic light sleep support will be disabled.

Default value:

- No (disabled) if *CONFIG_PM_ENABLE*

CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP

configEXPECTED_IDLE_TIME_BEFORE_SLEEP

Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_USE_TICKLESS_IDLE

FreeRTOS will enter light sleep mode if no tasks need to run for this number of ticks.

Range:

- from 2 to 4294967295 if *CONFIG_FREERTOS_USE_TICKLESS_IDLE*

Default value:

- 3 if *CONFIG_FREERTOS_USE_TICKLESS_IDLE*

Port Contains:

- *CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER*
- *CONFIG_FREERTOS_RUN_TIME_STATS_CLK*
- *CONFIG_FREERTOS_INTERRUPT_BACKTRACE*
- *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK*
- *CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP*
- *CONFIG_FREERTOS_ENABLE_TASK_SNAPSHOT*
- *CONFIG_FREERTOS_TLSP_DELETION_CALLBACKS*
- *CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION*
- *CONFIG_FREERTOS_ISR_STACKSIZE*
- *CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH*
- *CONFIG_FREERTOS_PLACE_SNAPSHOT_FUNS_INTO_FLASH*

- *CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE*
- *CONFIG_FREERTOS_CORETIMER*
- *CONFIG_FREERTOS_FPU_IN_ISR*
- *CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER*

CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER

Wrap task functions

Found in: Component config > FreeRTOS > Port

If enabled, all FreeRTOS task functions will be enclosed in a wrapper function. If a task function mistakenly returns (i.e. does not delete), the call flow will return to the wrapper function. The wrapper function will then log an error and abort the application. This option is also required for GDB backtraces and C++ exceptions to work correctly inside top-level task functions.

Default value:

- Yes (enabled)

CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK

Enable stack overflow debug watchpoint

Found in: Component config > FreeRTOS > Port

FreeRTOS can check if a stack has overflown its bounds by checking either the value of the stack pointer or by checking the integrity of canary bytes. (See FREERTOS_CHECK_STACKOVERFLOW for more information.) These checks only happen on a context switch, and the situation that caused the stack overflow may already be long gone by then. This option will use the last debug memory watchpoint to allow breaking into the debugger (or panic'ing) as soon as any of the last 32 bytes on the stack of a task are overwritten. The side effect is that using gdb, you effectively have one hardware watchpoint less because the last one is overwritten as soon as a task switch happens.

Another consequence is that due to alignment requirements of the watchpoint, the usable stack size decreases by up to 60 bytes. This is because the watchpoint region has to be aligned to its size and the size for the stack watchpoint in IDF is 32 bytes.

This check only triggers if the stack overflow writes within 32 bytes near the end of the stack, rather than overshooting further, so it is worth combining this approach with one of the other stack overflow check methods.

When this watchpoint is hit, gdb will stop with a SIGTRAP message. When no JTAG OCD is attached, esp-idf will panic on an unhandled debug exception.

Default value:

- No (disabled)

CONFIG_FREERTOS_TLSP_DELETION_CALLBACKS

Enable thread local storage pointers deletion callbacks

Found in: Component config > FreeRTOS > Port

ESP-IDF provides users with the ability to free TLSP memory by registering TLSP deletion callbacks. These callbacks are automatically called by FreeRTOS when a task is deleted. When this option is turned on, the memory reserved for TLSPs in the TCB is doubled to make space for storing the deletion callbacks. If the user does not wish to use TLSP deletion callbacks then this option could be turned off to save space in the TCB memory.

Default value:

- Yes (enabled) if *CONFIG_FREERTOS_SMP* && *CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS* > 0

CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP

Enable static task clean up hook

Found in: Component config > FreeRTOS > Port

Enable this option to make FreeRTOS call the static task clean up hook when a task is deleted.

Note: Users will need to provide a `void vPortCleanUpTCB ( void \*pxTCB )` callback

Default value:

- No (disabled) if *CONFIG_FREERTOS_SMP*

CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER

Check that mutex semaphore is given by owner task

Found in: Component config > FreeRTOS > Port

If enabled, assert that when a mutex semaphore is given, the task giving the semaphore is the task which is currently holding the mutex.

Default value:

- Yes (enabled)

CONFIG_FREERTOS_ISR_STACKSIZE

ISR stack size

Found in: Component config > FreeRTOS > Port

The interrupt handlers have their own stack. The size of the stack can be defined here. Each processor has its own stack, so the total size occupied will be twice this.

Range:

- from 2096 to 32768 if ESP_COREDUMP_DATA_FORMAT_ELF
- from 1536 to 32768

Default value:

- 2096 if ESP_COREDUMP_DATA_FORMAT_ELF
- 1536

CONFIG_FREERTOS_INTERRUPT_BACKTRACE

Enable backtrace from interrupt to task context

Found in: Component config > FreeRTOS > Port

If this option is enabled, interrupt stack frame will be modified to point to the code of the interrupted task as its return address. This helps the debugger (or the panic handler) show a backtrace from the interrupt to the task which was interrupted. This also works for nested interrupts: higher level interrupt stack can be traced back to the lower level interrupt. This option adds 4 instructions to the interrupt dispatching code.

Default value:

- Yes (enabled)

CONFIG_FREERTOS_FPU_IN_ISR

Use float in Level 1 ISR

Found in: Component config > FreeRTOS > Port

When enabled, the usage of float type is allowed inside Level 1 ISRs. Note that usage of float types in higher level interrupts is still not permitted.

Default value:

- No (disabled)

CONFIG_FREERTOS_CORETIMER

Tick timer source (Xtensa Only)

Found in: Component config > FreeRTOS > Port

FreeRTOS needs a timer with an associated interrupt to use as the main tick source to increase counters, run timers and do pre-emptive multitasking with. There are multiple timers available to do this, with different interrupt priorities.

Available options:

- Timer 0 (int 6, level 1) (FREERTOS_CORETIMER_0)
  Select this to use timer 0
- Timer 1 (int 15, level 3) (FREERTOS_CORETIMER_1)
  Select this to use timer 1
- SYSTIMER 0 (level 1) (FREERTOS_CORETIMER_SYSTIMER_LVL1)
  Select this to use systimer with the 1 interrupt priority.
- SYSTIMER 0 (level 3) (FREERTOS_CORETIMER_SYSTIMER_LVL3)
  Select this to use systimer with the 3 interrupt priority.

CONFIG_FREERTOS_RUN_TIME_STATS_CLK

Choose the clock source for run time stats

Found in: Component config > FreeRTOS > Port

Choose the clock source for FreeRTOS run time stats. Options are CPU0’ s CPU Clock or the ESP Timer. Both clock sources are 32 bits. The CPU Clock can run at a higher frequency hence provide a finer resolution but will overflow much quicker. Note that run time stats are only valid until the clock source overflows.

Available options:

- Use ESP TIMER for run time stats (FREERTOS_RUN_TIME_STATS_USING_ESP_TIMER)
  ESP Timer will be used as the clock source for FreeRTOS run time stats. The ESP Timer runs at a frequency of 1MHz regardless of Dynamic Frequency Scaling. Therefore the ESP Timer will overflow in approximately 4290 seconds.
- Use CPU Clock for run time stats (FREERTOS_RUN_TIME_STATS_USING_CPU_CLK)
  CPU Clock will be used as the clock source for the generation of run time stats. The CPU Clock has a frequency dependent on ESP_DEFAULT_CPU_FREQ_MHZ and Dynamic Frequency Scaling (DFS). Therefore the CPU Clock frequency can fluctuate between 80 to 240MHz. Run time stats generated using the CPU Clock represents the number of CPU cycles each task is allocated and DOES NOT reflect the amount of time each task runs for (as CPU clock frequency can change). If the CPU clock consistently runs at the maximum frequency of 240MHz, it will overflow in approximately 17 seconds.

CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH

Place FreeRTOS functions into Flash

Found in: Component config > FreeRTOS > Port

When enabled the selected Non-ISR FreeRTOS functions will be placed into Flash memory instead of IRAM. This saves up to 8KB of IRAM depending on which functions are used.

Default value:

- No (disabled)

CONFIG_FREERTOS_PLACE_SNAPSHOT_FUNS_INTO_FLASH

Place task snapshot functions into flash

Found in: Component config > FreeRTOS > Port

When enabled, the functions related to snapshots, such as vTaskGetSnapshot or uxTaskGetSnapshotAll, will be placed in flash. Note that if enabled, these functions cannot be called when cache is disabled.

Default value:

- No (disabled) if *CONFIG_FREERTOS_ENABLE_TASK_SNAPSHOT* && *CONFIG_ESP_PANIC_HANDLER_IRAM*

CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE

Tests compliance with Vanilla FreeRTOS port*_CRITICAL calls

Found in: Component config > FreeRTOS > Port

If enabled, context of port*_CRITICAL calls (ISR or Non-ISR) would be checked to be in compliance with Vanilla FreeRTOS. e.g Calling port*_CRITICAL from ISR context would cause assert failure

Default value:

- No (disabled)

CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION

Halt when an SMP-untested function is called

Found in: Component config > FreeRTOS > Port

Some functions in FreeRTOS have not been thoroughly tested yet when moving to the SMP implementation of FreeRTOS. When this option is enabled, these functions will throw an `assert()`.

Default value:

- Yes (enabled)

CONFIG_FREERTOS_ENABLE_TASK_SNAPSHOT

Enable task snapshot functions

Found in: Component config > FreeRTOS > Port

When enabled, the functions related to snapshots, such as vTaskGetSnapshot or uxTaskGetSnapshotAll, are compiled and linked. Task snapshots are used by Task Watchdog (TWDT), GDB Stub and Core dump.

Default value:

- Yes (enabled)

Hardware Abstraction Layer (HAL) and Low Level (LL)

Contains:

- *CONFIG_HAL_DEFAULT_ASSERTION_LEVEL*
- *CONFIG_HAL_LOG_LEVEL*
- *CONFIG_HAL_SYSTIMER_USE_ROM_IMPL*
- *CONFIG_HAL_WDT_USE_ROM_IMPL*

CONFIG_HAL_DEFAULT_ASSERTION_LEVEL

Default HAL assertion level

Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)

Set the assert behavior / level for HAL component. HAL component assert level can be set separately, but the level can't exceed the system assertion level. e.g. If the system assertion is disabled, then the

HAL assertion can't be enabled either. If the system assertion is enable, then the HAL assertion can still be disabled by this Kconfig option.

Available options:

- Same as system assertion level (HAL_ASSERTION_EQUALS_SYSTEM)
- Disabled (HAL_ASSERTION_DISABLE)
- Silent (HAL_ASSERTION_SILENT)
- Enabled (HAL_ASSERTION_ENABLE)

CONFIG_HAL_LOG_LEVEL

HAL layer log verbosity

Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)

Specify how much output to see in HAL logs.

Available options:

- No output (HAL_LOG_LEVEL_NONE)
- Error (HAL_LOG_LEVEL_ERROR)
- Warning (HAL_LOG_LEVEL_WARN)
- Info (HAL_LOG_LEVEL_INFO)
- Debug (HAL_LOG_LEVEL_DEBUG)
- Verbose (HAL_LOG_LEVEL_VERBOSE)

CONFIG_HAL_SYSTIMER_USE_ROM_IMPL

Use ROM implementation of SysTimer HAL driver

Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)

Enable this flag to use HAL functions from ROM instead of ESP-IDF.

If keeping this as "n" in your project, you will have less free IRAM. If making this as "y" in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

Default value:

- Yes (enabled) if ESP_ROM_HAS_HAL_SYSTIMER

CONFIG_HAL_WDT_USE_ROM_IMPL

Use ROM implementation of WDT HAL driver

Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)

Enable this flag to use HAL functions from ROM instead of ESP-IDF.

If keeping this as "n" in your project, you will have less free IRAM. If making this as "y" in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

Default value:

- Yes (enabled) if ESP_ROM_HAS_HAL_WDT

Heap memory debugging

Contains:

- *CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS*
- *CONFIG_HEAP_TASK_TRACKING*
- *CONFIG_HEAP_CORRUPTION_DETECTION*
- *CONFIG_HEAP_TRACING_DEST*
- *CONFIG_HEAP_TRACING_STACK_DEPTH*
- *CONFIG_HEAP_TLSF_USE_ROM_IMPL*

CONFIG_HEAP_CORRUPTION_DETECTION

Heap corruption detection

Found in: Component config > Heap memory debugging

Enable heap poisoning features to detect heap corruption caused by out-of-bounds access to heap memory.

See the "Heap Memory Debugging" page of the IDF documentation for a description of each level of heap corruption detection.

Available options:

- Basic (no poisoning) (HEAP_POISONING_DISABLED)
- Light impact (HEAP_POISONING_LIGHT)
- Comprehensive (HEAP_POISONING_COMPREHENSIVE)

CONFIG_HEAP_TRACING_DEST

Heap tracing

Found in: Component config > Heap memory debugging

Enables the heap tracing API defined in esp_heap_trace.h.

This function causes a moderate increase in IRAM code side and a minor increase in heap function (malloc/free/realloc) CPU overhead, even when the tracing feature is not used. So it's best to keep it disabled unless tracing is being used.

Available options:

- Disabled (HEAP_TRACING_OFF)
- Standalone (HEAP_TRACING_STANDALONE)
- Host-based (HEAP_TRACING_TOHOST)

CONFIG_HEAP_TRACING_STACK_DEPTH

Heap tracing stack depth

Found in: Component config > Heap memory debugging

Number of stack frames to save when tracing heap operation callers.

More stack frames uses more memory in the heap trace buffer (and slows down allocation), but can provide useful information.

CONFIG_HEAP_TASK_TRACKING

Enable heap task tracking

Found in: Component config > Heap memory debugging

Enables tracking the task responsible for each heap allocation.

This function depends on heap poisoning being enabled and adds four more bytes of overhead for each block allocated.

CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS

Abort if memory allocation fails

Found in: Component config > Heap memory debugging

When enabled, if a memory allocation operation fails it will cause a system abort.

Default value:

- No (disabled)

CONFIG_HEAP_TLSF_USE_ROM_IMPL

Use ROM implementation of heap tlsf library

Found in: Component config > Heap memory debugging

Enable this flag to use heap functions from ROM instead of ESP-IDF.

If keeping this as "n" in your project, you will have less free IRAM. If making this as "y" in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

Default value:

- Yes (enabled) if ESP_ROM_HAS_HEAP_TLSF

Log output Contains:

- *CONFIG_LOG_DEFAULT_LEVEL*
- *CONFIG_LOG_TIMESTAMP_SOURCE*
- *CONFIG_LOG_MAXIMUM_LEVEL*
- *CONFIG_LOG_COLORS*

CONFIG_LOG_DEFAULT_LEVEL

Default log verbosity

Found in: Component config > Log output

Specify how much output to see in logs by default. You can set lower verbosity level at runtime using esp_log_level_set function.

By default, this setting limits which log statements are compiled into the program. For example, selecting "Warning" would mean that changing log level to "Debug" at runtime will not be possible. To allow increasing log level above the default at runtime, see the next option.

Available options:

- No output (LOG_DEFAULT_LEVEL_NONE)
- Error (LOG_DEFAULT_LEVEL_ERROR)
- Warning (LOG_DEFAULT_LEVEL_WARN)
- Info (LOG_DEFAULT_LEVEL_INFO)
- Debug (LOG_DEFAULT_LEVEL_DEBUG)
- Verbose (LOG_DEFAULT_LEVEL_VERBOSE)

CONFIG_LOG_MAXIMUM_LEVEL

Maximum log verbosity

Found in: Component config > Log output

This config option sets the highest log verbosity that it's possible to select at runtime by calling esp_log_level_set(). This level may be higher than the default verbosity level which is set when the app starts up.

This can be used enable debugging output only at a critical point, for a particular tag, or to minimize startup time but then enable more logs once the firmware has loaded.

Note that increasing the maximum available log level will increase the firmware binary size.

This option only applies to logging from the app, the bootloader log level is fixed at compile time to the separate "Bootloader log verbosity" setting.

Available options:

- Same as default (LOG_MAXIMUM_EQUALS_DEFAULT)
- Error (LOG_MAXIMUM_LEVEL_ERROR)
- Warning (LOG_MAXIMUM_LEVEL_WARN)

- Info (LOG_MAXIMUM_LEVEL_INFO)
- Debug (LOG_MAXIMUM_LEVEL_DEBUG)
- Verbose (LOG_MAXIMUM_LEVEL_VERBOSE)

CONFIG_LOG_COLORS

Use ANSI terminal colors in log output

Found in: Component config > Log output

Enable ANSI terminal color codes in bootloader output.

In order to view these, your terminal program must support ANSI color codes.

Default value:

- Yes (enabled)

CONFIG_LOG_TIMESTAMP_SOURCE

Log Timestamps

Found in: Component config > Log output

Choose what sort of timestamp is displayed in the log output:

- Milliseconds since boot is calulated from the RTOS tick count multiplied by the tick period. This time will reset after a software reboot. e.g. (90000)
- System time is taken from POSIX time functions which use the chip's RTC and high resoultion timers to maintain an accurate time. The system time is initialized to 0 on startup, it can be set with an SNTP sync, or with POSIX time functions. This time will not reset after a software reboot. e.g. (00:01:30.000)
- NOTE: Currently this will not get used in logging from binary blobs (i.e WiFi & Bluetooth libraries), these will always print milliseconds since boot.

Available options:

- Milliseconds Since Boot (LOG_TIMESTAMP_SOURCE_RTOS)
- System Time (LOG_TIMESTAMP_SOURCE_SYSTEM)

LWIP

Contains:

- *CONFIG_LWIP_CHECK_THREAD_SAFETY*
- *Checksums*
- *CONFIG_LWIP_DHCP_COARSE_TIMER_SECS*
- *DHCP server*
- *CONFIG_LWIP_DHCP_OPTIONS_LEN*
- *CONFIG_LWIP_DHCP_DISABLE_CLIENT_ID*
- *CONFIG_LWIP_DHCP_DISABLE_VENDOR_CLASS_ID*
- *CONFIG_LWIP_DHCP_DOES_ARP_CHECK*
- *CONFIG_LWIP_DHCP_RESTORE_LAST_IP*
- *DNS*
- *CONFIG_LWIP_PPP_CHAP_SUPPORT*
- *CONFIG_LWIP_L2_TO_L3_COPY*
- *CONFIG_LWIP_IPV6_DHCP6*
- *CONFIG_LWIP_IP4_FRAG*
- *CONFIG_LWIP_IP6_FRAG*
- *CONFIG_LWIP_IP_FORWARD*
- *CONFIG_LWIP_NETBUF_RECVINFO*
- *CONFIG_LWIP_AUTOIP*
- *CONFIG_LWIP_IPV6*
- *CONFIG_LWIP_ENABLE_LCP_ECHO*
- *CONFIG_LWIP_ESP_LWIP_ASSERT*

- *CONFIG_LWIP_DEBUG*
- *CONFIG_LWIP_IRAM_OPTIMIZATION*
- *CONFIG_LWIP_STATS*
- *CONFIG_LWIP_TIMERS_ONDEMAND*
- *CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES*
- *CONFIG_LWIP_PPP_MPPE_SUPPORT*
- *CONFIG_LWIP_PPP_MSCHAP_SUPPORT*
- *CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT*
- *CONFIG_LWIP_PPP_PAP_SUPPORT*
- *CONFIG_LWIP_PPP_DEBUG_ON*
- *CONFIG_LWIP_PPP_SUPPORT*
- *CONFIG_LWIP_IP4_REASSEMBLY*
- *CONFIG_LWIP_IP6_REASSEMBLY*
- *CONFIG_LWIP_SLIP_SUPPORT*
- *CONFIG_LWIP_SO_LINGER*
- *CONFIG_LWIP_SO_RCVBUF*
- *CONFIG_LWIP_SO_REUSE*
- *CONFIG_LWIP_NETIF_STATUS_CALLBACK*
- *CONFIG_LWIP_TCPIP_CORE_LOCKING*
- *CONFIG_LWIP_NETIF_API*
- *Hooks*
- *ICMP*
- *CONFIG_LWIP_LOCAL_HOSTNAME*
- *CONFIG_LWIP_ND6*
- *LWIP RAW API*
- *CONFIG_LWIP_TCPIP_TASK_PRIO*
- *CONFIG_LWIP_IPV6_ND6_NUM_NEIGHBORS*
- *CONFIG_LWIP_IPV6_MEMP_NUM_ND6_QUEUE*
- *CONFIG_LWIP_MAX_SOCKETS*
- *CONFIG_LWIP_BRIDGEIF_MAX_PORTS*
- *CONFIG_LWIP_NUM_NETIF_CLIENT_DATA*
- *CONFIG_LWIP_ESP_GRATUITOUS_ARP*
- *CONFIG_LWIP_ESP_MLDV6_REPORT*
- *SNTP*
- *CONFIG_LWIP_USE_ONLY_LWIP_SELECT*
- *CONFIG_LWIP_NETIF_LOOPBACK*
- *TCP*
- *CONFIG_LWIP_TCPIP_TASK_AFFINITY*
- *CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*
- *CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*
- *CONFIG_LWIP_IP_DEFAULT_TTL*
- *UDP*
- *CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS*

CONFIG_LWIP_LOCAL_HOSTNAME

Local netif hostname

Found in: Component config > LWIP

The default name this device will report to other devices on the network. Could be updated at runtime with esp_netif_set_hostname()

Default value:

- "espressif"

CONFIG_LWIP_NETIF_API

Enable usage of standard POSIX APIs in LWIP

Found in: Component config > LWIP

If this feature is enabled, standard POSIX APIs: if_indextoname(), if_nametoindex() could be used to convert network interface index to name instead of IDF specific esp-netif APIs (such as esp_netif_get_netif_impl_name())

Default value:

- No (disabled)

CONFIG_LWIP_TCPIP_TASK_PRIO

LWIP TCP/IP Task Priority

Found in: Component config > LWIP

LWIP tcpip task priority. In case of high throughput, this parameter could be changed up to (config-MAX_PRIORITIES-1).

Range:

- from 1 to 24

Default value:

- 18

CONFIG_LWIP_TCPIP_CORE_LOCKING

Enable tcpip core locking

Found in: Component config > LWIP

If Enable tcpip core locking,Creates a global mutex that is held during TCPIP thread operations.Can be locked by client code to perform lwIP operations without changing into TCPIP thread using callbacks. See LOCK_TCPIP_CORE() and UNLOCK_TCPIP_CORE().

If disable tcpip core locking,TCP IP will perform tasks through context switching

Default value:

- No (disabled)

CONFIG_LWIP_TCPIP_CORE_LOCKING_INPUT

Enable tcpip core locking input

Found in: Component config > LWIP > CONFIG_LWIP_TCPIP_CORE_LOCKING

when LWIP_TCPIP_CORE_LOCKING is enabled, this lets tcpip_input() grab the mutex for input packets as well, instead of allocating a message and passing it to tcpip_thread.

Default value:

- No (disabled) if *CONFIG_LWIP_TCPIP_CORE_LOCKING*

CONFIG_LWIP_CHECK_THREAD_SAFETY

Checks that lwip API runs in expected context

Found in: Component config > LWIP

Enable to check that the project does not violate lwip thread safety. If enabled, all lwip functions that require thread awareness run an assertion to verify that the TCP/IP core functionality is either locked or accessed from the correct thread.

Default value:

- No (disabled)

CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES

Enable mDNS queries in resolving host name

Found in: Component config > LWIP

If this feature is enabled, standard API such as gethostbyname support .local addresses by sending one shot multicast mDNS query

Default value:

- Yes (enabled)

CONFIG_LWIP_L2_TO_L3_COPY

Enable copy between Layer2 and Layer3 packets

Found in: Component config > LWIP

If this feature is enabled, all traffic from layer2(WIFI Driver) will be copied to a new buffer before sending it to layer3(LWIP stack), freeing the layer2 buffer. Please be notified that the total layer2 receiving buffer is fixed and ESP32 currently supports 25 layer2 receiving buffer, when layer2 buffer runs out of memory, then the incoming packets will be dropped in hardware. The layer3 buffer is allocated from the heap, so the total layer3 receiving buffer depends on the available heap size, when heap runs out of memory, no copy will be sent to layer3 and packet will be dropped in layer2. Please make sure you fully understand the impact of this feature before enabling it.

Default value:

- No (disabled)

CONFIG_LWIP_IRAM_OPTIMIZATION

Enable LWIP IRAM optimization

Found in: Component config > LWIP

If this feature is enabled, some functions relating to RX/TX in LWIP will be put into IRAM, it can improve UDP/TCP throughput by >10% for single core mode, it doesn't help too much for dual core mode. On the other hand, it needs about 10KB IRAM for these optimizations.

If this feature is disabled, all lwip functions will be put into FLASH.

Default value:

- No (disabled)

CONFIG_LWIP_TIMERS_ONDEMAND

Enable LWIP Timers on demand

Found in: Component config > LWIP

If this feature is enabled, IGMP and MLD6 timers will be activated only when joining groups or receiving QUERY packets.

This feature will reduce the power consumption for applications which do not use IGMP and MLD6.

Default value:

- Yes (enabled)

CONFIG_LWIP_MAX_SOCKETS

Max number of open sockets

Found in: Component config > LWIP

Sockets take up a certain amount of memory, and allowing fewer sockets to be open at the same time conserves memory. Specify the maximum amount of sockets here. The valid value is from 1 to 16.

Range:
- from 1 to 16

Default value:
- 10

CONFIG_LWIP_USE_ONLY_LWIP_SELECT

Support LWIP socket select() only (DEPRECATED)

Found in: Component config > LWIP

This option is deprecated. Do not use this option, use VFS_SUPPORT_SELECT instead.

Default value:
- No (disabled)

CONFIG_LWIP_SO_LINGER

Enable SO_LINGER processing

Found in: Component config > LWIP

Enabling this option allows SO_LINGER processing. l_onoff = 1,l_linger can set the timeout.

If l_linger=0, When a connection is closed, TCP will terminate the connection. This means that TCP will discard any data packets stored in the socket send buffer and send an RST to the peer.

If l_linger!=0,Then closesocket() calls to block the process until the remaining data packets has been sent or timed out.

Default value:
- No (disabled)

CONFIG_LWIP_SO_REUSE

Enable SO_REUSEADDR option

Found in: Component config > LWIP

Enabling this option allows binding to a port which remains in TIME_WAIT.

Default value:
- Yes (enabled)

CONFIG_LWIP_SO_REUSE_RXTOALL

SO_REUSEADDR copies broadcast/multicast to all matches

Found in: Component config > LWIP > CONFIG_LWIP_SO_REUSE

Enabling this option means that any incoming broadcast or multicast packet will be copied to all of the local sockets that it matches (may be more than one if SO_REUSEADDR is set on the socket.)

This increases memory overhead as the packets need to be copied, however they are only copied per matching socket. You can safely disable it if you don’ t plan to receive broadcast or multicast traffic on more than one socket at a time.

Default value:
- Yes (enabled)

CONFIG_LWIP_SO_RCVBUF

Enable SO_RCVBUF option

Found in: Component config > LWIP

Enabling this option allows checking for available data on a netconn.

Default value:

- No (disabled)

CONFIG_LWIP_NETBUF_RECVINFO

Enable IP_PKTINFO option

Found in: Component config > LWIP

Enabling this option allows checking for the destination address of a received IPv4 Packet.

Default value:

- No (disabled)

CONFIG_LWIP_IP_DEFAULT_TTL

The value for Time-To-Live used by transport layers

Found in: Component config > LWIP

Set value for Time-To-Live used by transport layers.

Range:

- from 1 to 255

Default value:

- 64

CONFIG_LWIP_IP4_FRAG

Enable fragment outgoing IP4 packets

Found in: Component config > LWIP

Enabling this option allows fragmenting outgoing IP4 packets if their size exceeds MTU.

Default value:

- Yes (enabled)

CONFIG_LWIP_IP6_FRAG

Enable fragment outgoing IP6 packets

Found in: Component config > LWIP

Enabling this option allows fragmenting outgoing IP6 packets if their size exceeds MTU.

Default value:

- Yes (enabled)

CONFIG_LWIP_IP4_REASSEMBLY

Enable reassembly incoming fragmented IP4 packets

Found in: Component config > LWIP

Enabling this option allows reassemblying incoming fragmented IP4 packets.

Default value:

- No (disabled)

CONFIG_LWIP_IP6_REASSEMBLY

Enable reassembly incoming fragmented IP6 packets

Found in: Component config > LWIP

Enabling this option allows reassemblying incoming fragmented IP6 packets.

Default value:

- No (disabled)

CONFIG_LWIP_IP_FORWARD

Enable IP forwarding

Found in: Component config > LWIP

Enabling this option allows packets forwarding across multiple interfaces.

Default value:

- No (disabled)

CONFIG_LWIP_IPV4_NAPT

Enable NAT (new/experimental)

Found in: Component config > LWIP > CONFIG_LWIP_IP_FORWARD

Enabling this option allows Network Address and Port Translation.

Default value:

- No (disabled) if *CONFIG_LWIP_IP_FORWARD*

CONFIG_LWIP_STATS

Enable LWIP statistics

Found in: Component config > LWIP

Enabling this option allows LWIP statistics

Default value:

- No (disabled)

CONFIG_LWIP_ESP_GRATUITOUS_ARP

Send gratuitous ARP periodically

Found in: Component config > LWIP

Enable this option allows to send gratuitous ARP periodically.

This option solve the compatibility issues.If the ARP table of the AP is old, and the AP doesn't send ARP request to update it's ARP table, this will lead to the STA sending IP packet fail. Thus we send gratuitous ARP periodically to let AP update it's ARP table.

Default value:

- Yes (enabled)

CONFIG_LWIP_GARP_TMR_INTERVAL

GARP timer interval(seconds)

Found in: Component config > LWIP > CONFIG_LWIP_ESP_GRATUITOUS_ARP

Set the timer interval for gratuitous ARP. The default value is 60s

Default value:

- 60

CONFIG_LWIP_ESP_MLDV6_REPORT

Send mldv6 report periodically

Found in: Component config > LWIP

Enable this option allows to send mldv6 report periodically.

This option solve the issue that failed to receive multicast data. Some routers fail to forward multicast packets. To solve this problem, send multicast mdlv6 report to routers regularly.

Default value:

- Yes (enabled)

CONFIG_LWIP_MLDV6_TMR_INTERVAL

mldv6 report timer interval(seconds)

Found in: Component config > LWIP > CONFIG_LWIP_ESP_MLDV6_REPORT

Set the timer interval for mldv6 report. The default value is 30s

Default value:

- 40

CONFIG_LWIP_TCPIP_RECVMBOX_SIZE

TCPIP task receive mail box size

Found in: Component config > LWIP

Set TCPIP task receive mail box size. Generally bigger value means higher throughput but more memory. The value should be bigger than UDP/TCP mail box size.

Range:

- from 6 to 64 if *CONFIG_LWIP_WND_SCALE*
- from 6 to 1024 if *CONFIG_LWIP_WND_SCALE*

Default value:

- 32

CONFIG_LWIP_DHCP_DOES_ARP_CHECK

DHCP: Perform ARP check on any offered address

Found in: Component config > LWIP

Enabling this option performs a check (via ARP request) if the offered IP address is not already in use by another host on the network.

Default value:

- Yes (enabled)

CONFIG_LWIP_DHCP_DISABLE_CLIENT_ID

DHCP: Disable Use of HW address as client identification

Found in: Component config > LWIP

This option could be used to disable DHCP client identification with its MAC address. (Client id is used by DHCP servers to uniquely identify clients and are included in the DHCP packets as an option 61) Set this option to "y" in order to exclude option 61 from DHCP packets.

Default value:

- No (disabled)

CONFIG_LWIP_DHCP_DISABLE_VENDOR_CLASS_ID

DHCP: Disable Use of vendor class identification

Found in: Component config > LWIP

This option could be used to disable DHCP client vendor class identification. Set this option to "y" in order to exclude option 60 from DHCP packets.

Default value:

- Yes (enabled)

CONFIG_LWIP_DHCP_RESTORE_LAST_IP

DHCP: Restore last IP obtained from DHCP server

Found in: Component config > LWIP

When this option is enabled, DHCP client tries to re-obtain last valid IP address obtained from DHCP server. Last valid DHCP configuration is stored in nvs and restored after reset/power-up. If IP is still available, there is no need for sending discovery message to DHCP server and save some time.

Default value:

- No (disabled)

CONFIG_LWIP_DHCP_OPTIONS_LEN

DHCP total option length

Found in: Component config > LWIP

Set total length of outgoing DHCP option msg. Generally bigger value means it can carry more options and values. If your code meets LWIP_ASSERT due to option value is too long. Please increase the LWIP_DHCP_OPTIONS_LEN value.

Range:

- from 68 to 255

Default value:

- 68
- 108

CONFIG_LWIP_NUM_NETIF_CLIENT_DATA

Number of clients store data in netif

Found in: Component config > LWIP

Number of clients that may store data in client_data member array of struct netif.

Range:

- from 0 to 256

Default value:

- 0

CONFIG_LWIP_DHCP_COARSE_TIMER_SECS

DHCP coarse timer interval(s)

Found in: Component config > LWIP

Set DHCP coarse interval in seconds. A higher value will be less precise but cost less power consumption.

Range:
- from 1 to 10

Default value:
- 1

DHCP server Contains:

- *CONFIG_LWIP_DHCPS*

CONFIG_LWIP_DHCPS

DHCPS: Enable IPv4 Dynamic Host Configuration Protocol Server (DHCPS)

Found in: Component config > LWIP > DHCP server

Enabling this option allows the device to run the DHCP server (to dynamically assign IPv4 addresses to clients).

Default value:
- Yes (enabled)

CONFIG_LWIP_DHCPS_LEASE_UNIT

Multiplier for lease time, in seconds

Found in: Component config > LWIP > DHCP server > CONFIG_LWIP_DHCPS

The DHCP server is calculating lease time multiplying the sent and received times by this number of seconds per unit. The default is 60, that equals one minute.

Range:
- from 1 to 3600

Default value:
- 60

CONFIG_LWIP_DHCPS_MAX_STATION_NUM

Maximum number of stations

Found in: Component config > LWIP > DHCP server > CONFIG_LWIP_DHCPS

The maximum number of DHCP clients that are connected to the server. After this number is exceeded, DHCP server removes of the oldest device from it' s address pool, without notification.

Range:
- from 1 to 64

Default value:
- 8

CONFIG_LWIP_AUTOIP

Enable IPV4 Link-Local Addressing (AUTOIP)

Found in: Component config > LWIP

Enabling this option allows the device to self-assign an address in the 169.256/16 range if none is assigned statically or via DHCP.

See RFC 3927.

Default value:
- No (disabled)

Contains:

- *CONFIG_LWIP_AUTOIP_TRIES*
- *CONFIG_LWIP_AUTOIP_MAX_CONFLICTS*
- *CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL*

CONFIG_LWIP_AUTOIP_TRIES

DHCP Probes before self-assigning IPv4 LL address

Found in: Component config > LWIP > CONFIG_LWIP_AUTOIP

DHCP client will send this many probes before self-assigning a link local address.

From LWIP help: "This can be set as low as 1 to get an AutoIP address very quickly, but you should be prepared to handle a changing IP address when DHCP overrides AutoIP." (In the case of ESP-IDF, this means multiple SYSTEM_EVENT_STA_GOT_IP events.)

Range:
- from 1 to 100 if *CONFIG_LWIP_AUTOIP*

Default value:
- 2 if *CONFIG_LWIP_AUTOIP*

CONFIG_LWIP_AUTOIP_MAX_CONFLICTS

Max IP conflicts before rate limiting

Found in: Component config > LWIP > CONFIG_LWIP_AUTOIP

If the AUTOIP functionality detects this many IP conflicts while self-assigning an address, it will go into a rate limited mode.

Range:
- from 1 to 100 if *CONFIG_LWIP_AUTOIP*

Default value:
- 9 if *CONFIG_LWIP_AUTOIP*

CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL

Rate limited interval (seconds)

Found in: Component config > LWIP > CONFIG_LWIP_AUTOIP

If rate limiting self-assignment requests, wait this long between each request.

Range:
- from 5 to 120 if *CONFIG_LWIP_AUTOIP*

Default value:
- 20 if *CONFIG_LWIP_AUTOIP*

CONFIG_LWIP_IPV6

Enable IPv6

Found in: Component config > LWIP

Enable IPv6 function. If not use IPv6 function, set this option to n. If disabling LWIP_IPV6 then some other components (coap and asio) will no longer be available.

Default value:
- Yes (enabled)

CONFIG_LWIP_IPV6_AUTOCONFIG

Enable IPV6 stateless address autoconfiguration (SLAAC)

Found in: Component config > LWIP > CONFIG_LWIP_IPV6

Enabling this option allows the devices to IPV6 stateless address autoconfiguration (SLAAC).

See RFC 4862.

Default value:

- No (disabled)

CONFIG_LWIP_IPV6_NUM_ADDRESSES

Number of IPv6 addresses on each network interface

Found in: Component config > LWIP > CONFIG_LWIP_IPV6

The maximum number of IPv6 addresses on each interface. Any additional addresses will be discarded.

Default value:

- 3

CONFIG_LWIP_IPV6_FORWARD

Enable IPv6 forwarding between interfaces

Found in: Component config > LWIP > CONFIG_LWIP_IPV6

Forwarding IPv6 packets between interfaces is only required when acting as a router.

Default value:

- No (disabled)

CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS

Use IPv6 Router Advertisement Recursive DNS Server Option

Found in: Component config > LWIP

Use IPv6 Router Advertisement Recursive DNS Server Option (as per RFC 6106) to copy a defined maximum number of DNS servers to the DNS module. Set this option to a number of desired DNS servers advertised in the RA protocol. This feature is disabled when set to 0.

Default value:

- 0 if *CONFIG_LWIP_IPV6_AUTOCONFIG*

CONFIG_LWIP_IPV6_DHCP6

Enable DHCPv6 stateless address autoconfiguration

Found in: Component config > LWIP

Enable DHCPv6 for IPv6 stateless address autoconfiguration. Note that the dhcpv6 client has to be started using dhcp6_enable_stateless(netif); Note that the stateful address autoconfiguration is not supported.

Default value:

- No (disabled) if *CONFIG_LWIP_IPV6_AUTOCONFIG*

CONFIG_LWIP_NETIF_STATUS_CALLBACK

Enable status callback for network interfaces

Found in: Component config > LWIP

Enable callbacks when the network interface is up/down and addresses are changed.

Default value:

- No (disabled)

CONFIG_LWIP_NETIF_LOOPBACK

Support per-interface loopback

Found in: Component config > LWIP

Enabling this option means that if a packet is sent with a destination address equal to the interface' s own IP address, it will "loop back" and be received by this interface. Disabling this option disables support of loopback interface in lwIP

Default value:

- Yes (enabled)

Contains:

- *CONFIG_LWIP_LOOPBACK_MAX_PBUFS*

CONFIG_LWIP_LOOPBACK_MAX_PBUFS

Max queued loopback packets per interface

Found in: Component config > LWIP > CONFIG_LWIP_NETIF_LOOPBACK

Configure the maximum number of packets which can be queued for loopback on a given interface. Reducing this number may cause packets to be dropped, but will avoid filling memory with queued packet data.

Range:

- from 0 to 16

Default value:

- 8

TCP

Contains:

- *CONFIG_LWIP_TCP_WND_DEFAULT*
- *CONFIG_LWIP_TCP_SND_BUF_DEFAULT*
- *CONFIG_LWIP_TCP_RECVMBOX_SIZE*
- *CONFIG_LWIP_TCP_RTO_TIME*
- *CONFIG_LWIP_MAX_ACTIVE_TCP*
- *CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT*
- *CONFIG_LWIP_MAX_LISTENING_TCP*
- *CONFIG_LWIP_TCP_MAXRTX*
- *CONFIG_LWIP_TCP_SYNMAXRTX*
- *CONFIG_LWIP_TCP_MSL*
- *CONFIG_LWIP_TCP_MSS*
- *CONFIG_LWIP_TCP_OVERSIZE*
- *CONFIG_LWIP_TCP_QUEUE_OOSEQ*
- *CONFIG_LWIP_WND_SCALE*
- *CONFIG_LWIP_TCP_HIGH_SPEED_RETRANSMISSION*
- *CONFIG_LWIP_TCP_TMR_INTERVAL*

CONFIG_LWIP_MAX_ACTIVE_TCP

Maximum active TCP Connections

Found in: Component config > LWIP > TCP

The maximum number of simultaneously active TCP connections. The practical maximum limit is determined by available heap memory at runtime.

Changing this value by itself does not substantially change the memory usage of LWIP, except for preventing new TCP connections after the limit is reached.

Range:
- from 1 to 1024

Default value:
- 16

CONFIG_LWIP_MAX_LISTENING_TCP

Maximum listening TCP Connections

Found in: Component config > LWIP > TCP

The maximum number of simultaneously listening TCP connections. The practical maximum limit is determined by available heap memory at runtime.

Changing this value by itself does not substantially change the memory usage of LWIP, except for preventing new listening TCP connections after the limit is reached.

Range:
- from 1 to 1024

Default value:
- 16

CONFIG_LWIP_TCP_HIGH_SPEED_RETRANSMISSION

TCP high speed retransmissions

Found in: Component config > LWIP > TCP

Speed up the TCP retransmission interval. If disabled, it is recommended to change the number of SYN retransmissions to 6, and TCP initial rto time to 3000.

Default value:
- Yes (enabled)

CONFIG_LWIP_TCP_MAXRTX

Maximum number of retransmissions of data segments

Found in: Component config > LWIP > TCP

Set maximum number of retransmissions of data segments.

Range:
- from 3 to 12

Default value:
- 12

CONFIG_LWIP_TCP_SYNMAXRTX

Maximum number of retransmissions of SYN segments

Found in: Component config > LWIP > TCP

Set maximum number of retransmissions of SYN segments.

Range:
- from 3 to 12

Default value:
- 6
- 12

CONFIG_LWIP_TCP_MSS

Maximum Segment Size (MSS)

Found in: Component config > LWIP > TCP

Set maximum segment size for TCP transmission.

Can be set lower to save RAM, the default value 1460(ipv4)/1440(ipv6) will give best throughput. IPv4 TCP_MSS Range: 576 <= TCP_MSS <= 1460 IPv6 TCP_MSS Range: 1220<= TCP_MSS <= 1440

Range:
- from 536 to 1460

Default value:
- 1440

CONFIG_LWIP_TCP_TMR_INTERVAL

TCP timer interval(ms)

Found in: Component config > LWIP > TCP

Set TCP timer interval in milliseconds.

Can be used to speed connections on bad networks. A lower value will redeliver unacked packets faster.

Default value:
- 250

CONFIG_LWIP_TCP_MSL

Maximum segment lifetime (MSL)

Found in: Component config > LWIP > TCP

Set maximum segment lifetime in milliseconds.

Default value:
- 60000

CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT

Maximum FIN segment lifetime

Found in: Component config > LWIP > TCP

Set maximum segment lifetime in milliseconds.

Default value:
- 20000

CONFIG_LWIP_TCP_SND_BUF_DEFAULT

Default send buffer size

Found in: Component config > LWIP > TCP

Set default send buffer size for new TCP sockets.

Per-socket send buffer size can be changed at runtime with lwip_setsockopt(s, TCP_SNDBUF, ⋯).

This value must be at least 2x the MSS size, and the default is 4x the default MSS size.

Setting a smaller default SNDBUF size can save some RAM, but will decrease performance.

Range:
- from 2440 to 65535 if *CONFIG_LWIP_WND_SCALE*
- from 2440 to 1024000 if *CONFIG_LWIP_WND_SCALE*

Default value:
- 5760

CONFIG_LWIP_TCP_WND_DEFAULT

Default receive window size

Found in: Component config > LWIP > TCP

Set default TCP receive window size for new TCP sockets.

Per-socket receive window size can be changed at runtime with lwip_setsockopt(s, TCP_WINDOW, ⋯).

Setting a smaller default receive window size can save some RAM, but will significantly decrease performance.

Range:
- from 2440 to 65535 if *CONFIG_LWIP_WND_SCALE*
- from 2440 to 1024000 if *CONFIG_LWIP_WND_SCALE*

Default value:
- 5760

CONFIG_LWIP_TCP_RECVMBOX_SIZE

Default TCP receive mail box size

Found in: Component config > LWIP > TCP

Set TCP receive mail box size. Generally bigger value means higher throughput but more memory. The recommended value is: LWIP_TCP_WND_DEFAULT/TCP_MSS + 2, e.g. if LWIP_TCP_WND_DEFAULT=14360, TCP_MSS=1436, then the recommended receive mail box size is (14360/1436 + 2) = 12.

TCP receive mail box is a per socket mail box, when the application receives packets from TCP socket, LWIP core firstly posts the packets to TCP receive mail box and the application then fetches the packets from mail box. It means LWIP can caches maximum LWIP_TCP_RECCVMBOX_SIZE packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is LWIP_TCP_RECCVMBOX_SIZE multiples the maximum TCP socket number. In other words, the bigger LWIP_TCP_RECVMBOX_SIZE means more memory. On the other hand, if the receiv mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the TCP receive mail box is big enough to avoid packet drop between LWIP core and application.

Range:
- from 6 to 64 if *CONFIG_LWIP_WND_SCALE*
- from 6 to 1024 if *CONFIG_LWIP_WND_SCALE*

Default value:
- 6

CONFIG_LWIP_TCP_QUEUE_OOSEQ

Queue incoming out-of-order segments

Found in: Component config > LWIP > TCP

Queue incoming out-of-order segments for later use.

Disable this option to save some RAM during TCP sessions, at the expense of increased retransmissions if segments arrive out of order.

Default value:

- Yes (enabled)

CONFIG_LWIP_TCP_OOSEQ_TIMEOUT

Timeout for each pbuf queued in TCP OOSEQ, in RTOs.

Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ

The timeout value is TCP_OOSEQ_TIMEOUT * RTO.

Range:

- from 1 to 30

Default value:

- 6

CONFIG_LWIP_TCP_OOSEQ_MAX_PBUFS

The maximum number of pbufs queued on OOSEQ per pcb

Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ

If LWIP_TCP_OOSEQ_MAX_PBUFS = 0, TCP will not control the number of OOSEQ pbufs.

In a poor network environment, many out-of-order tcp pbufs will be received. These out-of-order pbufs will be cached in the TCP out-of-order queue which will cause Wi-Fi/Ethernet fail to release RX buffer in time. It is possible that all RX buffers for MAC layer are used by OOSEQ.

Control the number of out-of-order pbufs to ensure that the MAC layer has enough RX buffer to receive packets.

In the Wi-Fi scenario,recommended OOSEQ PBUFS Range: 0 <= TCP_OOSEQ_MAX_PBUFS <= CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM/(MAX_TCP_NUMBER + 1)

In the Ethernet scenario,recommended Ethernet OOSEQ PBUFS Range: 0 <= TCP_OOSEQ_MAX_PBUFS <= CONFIG_ETH_DMA_RX_BUFFER_NUM/(MAX_TCP_NUMBER + 1)

Within the recommended value range, the larger the value, the better the performance.

MAX_TCP_NUMBER represent Maximum number of TCP connections in Wi-Fi(STA+SoftAP) and Ethernet scenario.

Range:

- from 0 to 12

Default value:

- 4 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP* && *CONFIG_LWIP_TCP_QUEUE_OOSEQ*
- 0 if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP* && *CONFIG_LWIP_TCP_QUEUE_OOSEQ*

CONFIG_LWIP_TCP_SACK_OUT

Support sending selective acknowledgements

Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ

TCP will support sending selective acknowledgements (SACKs).

Default value:

- No (disabled)

CONFIG_LWIP_TCP_OVERSIZE

Pre-allocate transmit PBUF size

Found in: Component config > LWIP > TCP

Allows enabling "oversize" allocation of TCP transmission pbufs ahead of time, which can reduce the length of pbuf chains used for transmission.

This will not make a difference to sockets where Nagle's algorithm is disabled.

Default value of MSS is fine for most applications, 25% MSS may save some RAM when only transmitting small amounts of data. Disabled will have worst performance and fragmentation characteristics, but uses least RAM overall.

Available options:

- MSS (LWIP_TCP_OVERSIZE_MSS)
- 25% MSS (LWIP_TCP_OVERSIZE_QUARTER_MSS)
- Disabled (LWIP_TCP_OVERSIZE_DISABLE)

CONFIG_LWIP_WND_SCALE

Support TCP window scale

Found in: Component config > LWIP > TCP

Enable this feature to support TCP window scaling.

Default value:

- No (disabled) if *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*

CONFIG_LWIP_TCP_RCV_SCALE

Set TCP receiving window scaling factor

Found in: Component config > LWIP > TCP > CONFIG_LWIP_WND_SCALE

Enable this feature to support TCP window scaling.

Range:

- from 0 to 14 if *CONFIG_LWIP_WND_SCALE*

Default value:

- 0 if *CONFIG_LWIP_WND_SCALE*

CONFIG_LWIP_TCP_RTO_TIME

Default TCP rto time

Found in: Component config > LWIP > TCP

Set default TCP rto time for a reasonable initial rto. In bad network environment, recommend set value of rto time to 1500.

Default value:

- 3000
- 1500

UDP

Contains:

- *CONFIG_LWIP_UDP_RECVMBOX_SIZE*
- *CONFIG_LWIP_MAX_UDP_PCBS*

CONFIG_LWIP_MAX_UDP_PCBS

Maximum active UDP control blocks

Found in: Component config > LWIP > UDP

The maximum number of active UDP "connections" (ie UDP sockets sending/receiving data). The practical maximum limit is determined by available heap memory at runtime.

Range:
- from 1 to 1024

Default value:
- 16

CONFIG_LWIP_UDP_RECVMBOX_SIZE

Default UDP receive mail box size

Found in: Component config > LWIP > UDP

Set UDP receive mail box size. The recommended value is 6.

UDP receive mail box is a per socket mail box, when the application receives packets from UDP socket, LWIP core firstly posts the packets to UDP receive mail box and the application then fetches the packets from mail box. It means LWIP can caches maximum UDP_RECCVMBOX_SIZE packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is UDP_RECCVMBOX_SIZE multiples the maximum UDP socket number. In other words, the bigger UDP_RECVMBOX_SIZE means more memory. On the other hand, if the receiv mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the UDP receive mail box is big enough to avoid packet drop between LWIP core and application.

Range:
- from 6 to 64

Default value:
- 6

Checksums

Contains:

- *CONFIG_LWIP_CHECKSUM_CHECK_ICMP*
- *CONFIG_LWIP_CHECKSUM_CHECK_IP*
- *CONFIG_LWIP_CHECKSUM_CHECK_UDP*

CONFIG_LWIP_CHECKSUM_CHECK_IP

Enable LWIP IP checksums

Found in: Component config > LWIP > Checksums

Enable checksum checking for received IP messages

Default value:
- No (disabled)

CONFIG_LWIP_CHECKSUM_CHECK_UDP

Enable LWIP UDP checksums

Found in: Component config > LWIP > Checksums

Enable checksum checking for received UDP messages

Default value:
- No (disabled)

CONFIG_LWIP_CHECKSUM_CHECK_ICMP

Enable LWIP ICMP checksums

Found in: Component config > LWIP > Checksums

Enable checksum checking for received ICMP messages

Default value:
- Yes (enabled)

CONFIG_LWIP_TCPIP_TASK_STACK_SIZE

TCP/IP Task Stack Size

Found in: Component config > LWIP

Configure TCP/IP task stack size, used by LWIP to process multi-threaded TCP/IP operations. Setting this stack too small will result in stack overflow crashes.

Range:
- from 2048 to 65536

Default value:
- 3072

CONFIG_LWIP_TCPIP_TASK_AFFINITY

TCP/IP task affinity

Found in: Component config > LWIP

Allows setting LwIP tasks affinity, i.e. whether the task is pinned to CPU0, pinned to CPU1, or allowed to run on any CPU. Currently this applies to "TCP/IP" task and "Ping" task.

Available options:
- No affinity (LWIP_TCPIP_TASK_AFFINITY_NO_AFFINITY)
- CPU0 (LWIP_TCPIP_TASK_AFFINITY_CPU0)
- CPU1 (LWIP_TCPIP_TASK_AFFINITY_CPU1)

CONFIG_LWIP_PPP_SUPPORT

Enable PPP support (new/experimental)

Found in: Component config > LWIP

Enable PPP stack. Now only PPP over serial is possible.

PPP over serial support is experimental and unsupported.

Default value:
- No (disabled)

Contains:

- *CONFIG_LWIP_PPP_ENABLE_IPV6*

CONFIG_LWIP_PPP_ENABLE_IPV6

Enable IPV6 support for PPP connections (IPV6CP)

Found in: Component config > LWIP > CONFIG_LWIP_PPP_SUPPORT

Enable IPV6 support in PPP for the local link between the DTE (processor) and DCE (modem). There are some modems which do not support the IPV6 addressing in the local link. If they are requested for IPV6CP negotiation, they may time out. This would in turn fail the configuration for the whole link. If your modem is not responding correctly to PPP Phase Network, try to disable IPV6 support.

Default value:

- Yes (enabled) if *CONFIG_LWIP_PPP_SUPPORT* && *CONFIG_LWIP_IPV6*

CONFIG_LWIP_IPV6_MEMP_NUM_ND6_QUEUE

Max number of IPv6 packets to queue during MAC resolution

Found in: Component config > LWIP

Config max number of IPv6 packets to queue during MAC resolution.

Range:

- from 3 to 20

Default value:

- 3

CONFIG_LWIP_IPV6_ND6_NUM_NEIGHBORS

Max number of entries in IPv6 neighbor cache

Found in: Component config > LWIP

Config max number of entries in IPv6 neighbor cache

Range:

- from 3 to 10

Default value:

- 5

CONFIG_LWIP_ND6

LWIP NDP6 Enable/Disable

Found in: Component config > LWIP

This option is used to disable the Network Discovery Protocol (NDP) if it is not required. Please use this option with caution, as the NDP is essential for IPv6 functionality within a local network.

Default value:

- Yes (enabled)

CONFIG_LWIP_FORCE_ROUTER_FORWARDING

LWIP Force Router Forwarding Enable/Disable

Found in: Component config > LWIP > CONFIG_LWIP_ND6

This option is used to set the the router flag for the NA packets. When enabled, the router flag in NA packet will always set to 1, otherwise, never set router flag for NA packets.

Default value:

- No (disabled)

CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT

Enable Notify Phase Callback

Found in: Component config > LWIP

Enable to set a callback which is called on change of the internal PPP state machine.

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_PPP_PAP_SUPPORT

Enable PAP support

Found in: Component config > LWIP

Enable Password Authentication Protocol (PAP) support

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_PPP_CHAP_SUPPORT

Enable CHAP support

Found in: Component config > LWIP

Enable Challenge Handshake Authentication Protocol (CHAP) support

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_PPP_MSCHAP_SUPPORT

Enable MSCHAP support

Found in: Component config > LWIP

Enable Microsoft version of the Challenge-Handshake Authentication Protocol (MSCHAP) support

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_PPP_MPPE_SUPPORT

Enable MPPE support

Found in: Component config > LWIP

Enable Microsoft Point-to-Point Encryption (MPPE) support

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_ENABLE_LCP_ECHO

Enable LCP ECHO

Found in: Component config > LWIP

Enable LCP echo keepalive requests

Default value:

- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_LCP_ECHOINTERVAL

Echo interval (s)

Found in: Component config > LWIP > CONFIG_LWIP_ENABLE_LCP_ECHO

Interval in seconds between keepalive LCP echo requests, 0 to disable.

Range:

- from 0 to 1000000 if *CONFIG_LWIP_ENABLE_LCP_ECHO*

Default value:

- 3 if *CONFIG_LWIP_ENABLE_LCP_ECHO*

CONFIG_LWIP_LCP_MAXECHOFAILS

Maximum echo failures

Found in: Component config > LWIP > CONFIG_LWIP_ENABLE_LCP_ECHO

Number of consecutive unanswered echo requests before failure is indicated.

Range:
- from 0 to 100000 if *CONFIG_LWIP_ENABLE_LCP_ECHO*

Default value:
- 3 if *CONFIG_LWIP_ENABLE_LCP_ECHO*

CONFIG_LWIP_PPP_DEBUG_ON

Enable PPP debug log output

Found in: Component config > LWIP

Enable PPP debug log output

Default value:
- No (disabled) if *CONFIG_LWIP_PPP_SUPPORT*

CONFIG_LWIP_SLIP_SUPPORT

Enable SLIP support (new/experimental)

Found in: Component config > LWIP

Enable SLIP stack. Now only SLIP over serial is possible.

SLIP over serial support is experimental and unsupported.

Default value:
- No (disabled)

Contains:

- *CONFIG_LWIP_SLIP_DEBUG_ON*

CONFIG_LWIP_SLIP_DEBUG_ON

Enable SLIP debug log output

Found in: Component config > LWIP > CONFIG_LWIP_SLIP_SUPPORT

Enable SLIP debug log output

Default value:
- No (disabled) if *CONFIG_LWIP_SLIP_SUPPORT*

ICMP

Contains:

- *CONFIG_LWIP_ICMP*
- *CONFIG_LWIP_BROADCAST_PING*
- *CONFIG_LWIP_MULTICAST_PING*

CONFIG_LWIP_ICMP

ICMP: Enable ICMP

Found in: Component config > LWIP > ICMP

Enable ICMP module for check network stability

Default value:

- Yes (enabled)

CONFIG_LWIP_MULTICAST_PING

Respond to multicast pings

Found in: Component config > LWIP > ICMP

Default value:
- No (disabled)

CONFIG_LWIP_BROADCAST_PING

Respond to broadcast pings

Found in: Component config > LWIP > ICMP

Default value:
- No (disabled)

LWIP RAW API Contains:

- *CONFIG_LWIP_MAX_RAW_PCBS*

CONFIG_LWIP_MAX_RAW_PCBS

Maximum LWIP RAW PCBs

Found in: Component config > LWIP > LWIP RAW API

The maximum number of simultaneously active LWIP RAW protocol control blocks. The practical maximum limit is determined by available heap memory at runtime.

Range:
- from 1 to 1024

Default value:
- 16

SNTP Contains:

- *CONFIG_LWIP_SNTP_MAX_SERVERS*
- *CONFIG_LWIP_SNTP_UPDATE_DELAY*
- *CONFIG_LWIP_DHCP_GET_NTP_SRV*

CONFIG_LWIP_SNTP_MAX_SERVERS

Maximum number of NTP servers

Found in: Component config > LWIP > SNTP

Set maximum number of NTP servers used by LwIP SNTP module. First argument of sntp_setserver/sntp_setservername functions is limited to this value.

Range:
- from 1 to 16

Default value:
- 1

CONFIG_LWIP_DHCP_GET_NTP_SRV

Request NTP servers from DHCP

Found in: Component config > LWIP > SNTP

If enabled, LWIP will add 'NTP' to Parameter-Request Option sent via DHCP-request. DHCP server might reply with an NTP server address in option 42. SNTP callback for such replies should be set accordingly (see sntp_servermode_dhcp() func.)

Default value:

- No (disabled)

CONFIG_LWIP_DHCP_MAX_NTP_SERVERS

Maximum number of NTP servers aquired via DHCP

Found in: Component config > LWIP > SNTP > CONFIG_LWIP_DHCP_GET_NTP_SRV

Set maximum number of NTP servers aquired via DHCP-offer. Should be less or equal to "Maximum number of NTP servers", any extra servers would be just ignored.

Range:

- from 1 to 16 if *CONFIG_LWIP_DHCP_GET_NTP_SRV*

Default value:

- 1 if *CONFIG_LWIP_DHCP_GET_NTP_SRV*

CONFIG_LWIP_SNTP_UPDATE_DELAY

Request interval to update time (ms)

Found in: Component config > LWIP > SNTP

This option allows you to set the time update period via SNTP. Default is 1 hour. Must not be below 15 seconds by specification. (SNTPv4 RFC 4330 enforces a minimum update time of 15 seconds).

Range:

- from 15000 to 4294967295

Default value:

- 3600000

CONFIG_LWIP_BRIDGEIF_MAX_PORTS

Maximum number of bridge ports

Found in: Component config > LWIP

Set maximum number of ports a bridge can consists of.

Range:

- from 1 to 63

Default value:

- 7

DNS

Contains:

- *CONFIG_LWIP_FALLBACK_DNS_SERVER_SUPPORT*
- *CONFIG_LWIP_DNS_MAX_SERVERS*

CONFIG_LWIP_DNS_MAX_SERVERS

Maximum number of DNS servers

Found in: Component config > LWIP > DNS

Set maximum number of DNS servers. If fallback DNS servers are supported, the number of DNS servers needs to be greater than or equal to 3.

Range:
- from 1 to 4

Default value:
- 3

CONFIG_LWIP_FALLBACK_DNS_SERVER_SUPPORT

Enable DNS fallback server support

Found in: Component config > LWIP > DNS

Enable this feature to support DNS fallback server.

Default value:
- No (disabled)

CONFIG_LWIP_FALLBACK_DNS_SERVER_ADDRESS

DNS fallback server address

Found in: Component config > LWIP > DNS > CONFIG_LWIP_FALLBACK_DNS_SERVER_SUPPORT

This option allows you to config dns fallback server address.

Default value:
- "114.114.114.114" if *CONFIG_LWIP_FALLBACK_DNS_SERVER_SUPPORT*

CONFIG_LWIP_ESP_LWIP_ASSERT

Enable LWIP ASSERT checks

Found in: Component config > LWIP

Enable this option keeps LWIP assertion checks enabled. It is recommended to keep this option enabled.

If asserts are disabled for the entire project, they are also disabled for LWIP and this option is ignored.

Default value:
- Yes (enabled) if COMPILER_OPTIMIZATION_ASSERTIONS_DISABLE

Hooks Contains:

- *CONFIG_LWIP_HOOK_ND6_GET_GW*
- *CONFIG_LWIP_HOOK_IP6_INPUT*
- *CONFIG_LWIP_HOOK_IP6_ROUTE*
- *CONFIG_LWIP_HOOK_NETCONN_EXTERNAL_RESOLVE*
- *CONFIG_LWIP_HOOK_TCP_ISN*

CONFIG_LWIP_HOOK_TCP_ISN

TCP ISN Hook

Found in: Component config > LWIP > Hooks

Enables to define a TCP ISN hook to randomize initial sequence number in TCP connection. The default TCP ISN algorithm used in IDF (standardized in RFC 6528) produces ISN by combining an MD5 of

the new TCP id and a stable secret with the current time. This is because the lwIP implementation (*tcp_next_iss*) is not very strong, as it does not take into consideration any platform specific entropy source.

Set to LWIP_HOOK_TCP_ISN_CUSTOM to provide custom implementation. Set to LWIP_HOOK_TCP_ISN_NONE to use lwIP implementation.

Available options:

- No hook declared (LWIP_HOOK_TCP_ISN_NONE)
- Default implementation (LWIP_HOOK_TCP_ISN_DEFAULT)
- Custom implementation (LWIP_HOOK_TCP_ISN_CUSTOM)

CONFIG_LWIP_HOOK_IP6_ROUTE

IPv6 route Hook

Found in: Component config > LWIP > Hooks

Enables custom IPv6 route hook. Setting this to "default" provides weak implementation stub that could be overwritten in application code. Setting this to "custom" provides hook's declaration only and expects the application to implement it.

Available options:

- No hook declared (LWIP_HOOK_IP6_ROUTE_NONE)
- Default (weak) implementation (LWIP_HOOK_IP6_ROUTE_DEFAULT)
- Custom implementation (LWIP_HOOK_IP6_ROUTE_CUSTOM)

CONFIG_LWIP_HOOK_ND6_GET_GW

IPv6 get gateway Hook

Found in: Component config > LWIP > Hooks

Enables custom IPv6 route hook. Setting this to "default" provides weak implementation stub that could be overwritten in application code. Setting this to "custom" provides hook's declaration only and expects the application to implement it.

Available options:

- No hook declared (LWIP_HOOK_ND6_GET_GW_NONE)
- Default (weak) implementation (LWIP_HOOK_ND6_GET_GW_DEFAULT)
- Custom implementation (LWIP_HOOK_ND6_GET_GW_CUSTOM)

CONFIG_LWIP_HOOK_NETCONN_EXTERNAL_RESOLVE

Netconn external resolve Hook

Found in: Component config > LWIP > Hooks

Enables custom DNS resolve hook. Setting this to "default" provides weak implementation stub that could be overwritten in application code. Setting this to "custom" provides hook's declaration only and expects the application to implement it.

Available options:

- No hook declared (LWIP_HOOK_NETCONN_EXT_RESOLVE_NONE)
- Default (weak) implementation (LWIP_HOOK_NETCONN_EXT_RESOLVE_DEFAULT)
- Custom implementation (LWIP_HOOK_NETCONN_EXT_RESOLVE_CUSTOM)

CONFIG_LWIP_HOOK_IP6_INPUT

IPv6 packet input

Found in: Component config > LWIP > Hooks

Enables custom IPv6 packet input. Setting this to "default" provides weak implementation stub that could be overwritten in application code. Setting this to "custom" provides hook's declaration only and expects the application to implement it.

Available options:

- No hook declared (LWIP_HOOK_IP6_INPUT_NONE)
- Default (weak) implementation (LWIP_HOOK_IP6_INPUT_DEFAULT)
- Custom implementation (LWIP_HOOK_IP6_INPUT_CUSTOM)

CONFIG_LWIP_DEBUG

Enable LWIP Debug

Found in: Component config > LWIP

Enabling this option allows different kinds of lwIP debug output.

All lwIP debug features increase the size of the final binary.

Default value:

- No (disabled)

Contains:

- *CONFIG_LWIP_API_LIB_DEBUG*
- *CONFIG_LWIP_BRIDGEIF_FDB_DEBUG*
- *CONFIG_LWIP_BRIDGEIF_FW_DEBUG*
- *CONFIG_LWIP_BRIDGEIF_DEBUG*
- *CONFIG_LWIP_DHCP_DEBUG*
- *CONFIG_LWIP_DHCP_STATE_DEBUG*
- *CONFIG_LWIP_DNS_DEBUG*
- *CONFIG_LWIP_ETHARP_DEBUG*
- *CONFIG_LWIP_ICMP_DEBUG*
- *CONFIG_LWIP_ICMP6_DEBUG*
- *CONFIG_LWIP_IP_DEBUG*
- *CONFIG_LWIP_IP6_DEBUG*
- *CONFIG_LWIP_NETIF_DEBUG*
- *CONFIG_LWIP_PBUF_DEBUG*
- *CONFIG_LWIP_SNTP_DEBUG*
- *CONFIG_LWIP_SOCKETS_DEBUG*
- *CONFIG_LWIP_TCP_DEBUG*
- *CONFIG_LWIP_DEBUG_ESP_LOG*

CONFIG_LWIP_DEBUG_ESP_LOG

Route LWIP debugs through ESP_LOG interface

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Enabling this option routes all enabled LWIP debugs through ESP_LOGD.

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_NETIF_DEBUG

Enable netif debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_PBUF_DEBUG

Enable pbuf debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_ETHARP_DEBUG

Enable etharp debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_API_LIB_DEBUG

Enable api lib debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_SOCKETS_DEBUG

Enable socket debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_IP_DEBUG

Enable IP debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_ICMP_DEBUG

Enable ICMP debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG* && *CONFIG_LWIP_ICMP*

CONFIG_LWIP_DHCP_STATE_DEBUG

Enable DHCP state tracking

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_DHCP_DEBUG

Enable DHCP debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_IP6_DEBUG

Enable IP6 debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_ICMP6_DEBUG

Enable ICMP6 debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_TCP_DEBUG

Enable TCP debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_SNTP_DEBUG

Enable SNTP debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_DNS_DEBUG

Enable DNS debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_BRIDGEIF_DEBUG

Enable bridge generic debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_BRIDGEIF_FDB_DEBUG

Enable bridge FDB debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

CONFIG_LWIP_BRIDGEIF_FW_DEBUG

Enable bridge forwarding debug messages

Found in: Component config > LWIP > CONFIG_LWIP_DEBUG

Default value:

- No (disabled) if *CONFIG_LWIP_DEBUG*

mbedTLS Contains:

- *CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN*
- *Certificate Bundle*
- *Certificates*
- *CONFIG_MBEDTLS_CHACHA20_C*
- *CONFIG_MBEDTLS_DHM_C*
- *CONFIG_MBEDTLS_ECP_C*
- *CONFIG_MBEDTLS_ECDH_C*
- *CONFIG_MBEDTLS_ECJPAKE_C*
- *CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED*
- *CONFIG_MBEDTLS_CMAC_C*
- *CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED*
- *CONFIG_MBEDTLS_ECDSA_DETERMINISTIC*
- *CONFIG_MBEDTLS_ECP_FIXED_POINT_OPTIM*
- *CONFIG_MBEDTLS_HARDWARE_AES*
- *CONFIG_MBEDTLS_HARDWARE_ECC*
- *CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN*
- *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*
- *CONFIG_MBEDTLS_HARDWARE_MPI*
- *CONFIG_MBEDTLS_HARDWARE_SHA*
- *CONFIG_MBEDTLS_DEBUG*
- *CONFIG_MBEDTLS_ECP_RESTARTABLE*
- *CONFIG_MBEDTLS_HAVE_TIME*
- *CONFIG_MBEDTLS_RIPEMD160_C*
- *CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED*
- *CONFIG_MBEDTLS_SHA512_C*
- *CONFIG_MBEDTLS_THREADING_C*
- *CONFIG_MBEDTLS_LARGE_KEY_SOFTWARE_MPI*
- *CONFIG_MBEDTLS_HKDF_C*
- *mbedTLS v3.x related*
- *CONFIG_MBEDTLS_MEM_ALLOC_MODE*
- *CONFIG_MBEDTLS_ECP_NIST_OPTIM*

- *CONFIG_MBEDTLS_POLY1305_C*
- *CONFIG_MBEDTLS_SSL_ALPN*
- *CONFIG_MBEDTLS_SSL_PROTO_DTLS*
- *CONFIG_MBEDTLS_SSL_PROTO_GMTSSL1_1*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_2*
- *CONFIG_MBEDTLS_SSL_RENEGOTIATION*
- *Symmetric Ciphers*
- *TLS Key Exchange Methods*
- *CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN*
- *CONFIG_MBEDTLS_TLS_MODE*
- *CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_ROM_MD5*
- *CONFIG_MBEDTLS_USE_CRYPTO_ROM_IMPL*
- *CONFIG_MBEDTLS_DYNAMIC_BUFFER*

CONFIG_MBEDTLS_MEM_ALLOC_MODE

Memory allocation strategy

Found in: Component config > mbedTLS

Allocation strategy for mbedTLS, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only
- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF
- Custom allocation mode, by overwriting calloc()/free() using mbedtls_platform_set_calloc_free() function
- Internal IRAM memory wherever applicable else internal DRAM

Recommended mode here is always internal (*), since that is most preferred from security perspective. But if application requirement does not allow sufficient free internal memory then alternate mode can be selected.

(*) In case of ESP32-S2/ESP32-S3, hardware allows encryption of external SPIRAM contents provided hardware flash encryption feature is enabled. In that case, using external SPIRAM allocation strategy is also safe choice from security perspective.

Available options:

- Internal memory (MBEDTLS_INTERNAL_MEM_ALLOC)
- External SPIRAM (MBEDTLS_EXTERNAL_MEM_ALLOC)
- Default alloc mode (MBEDTLS_DEFAULT_MEM_ALLOC)
- Custom alloc mode (MBEDTLS_CUSTOM_MEM_ALLOC)
- Internal IRAM (MBEDTLS_IRAM_8BIT_MEM_ALLOC)
  Allows to use IRAM memory region as 8bit accessible region.
  TLS input and output buffers will be allocated in IRAM section which is 32bit aligned memory. Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN

TLS maximum message content length

Found in: Component config > mbedTLS

Maximum TLS message length (in bytes) supported by mbedTLS.

16384 is the default and this value is required to comply fully with TLS standards.

However you can set a lower value in order to save RAM. This is safe if the other end of the connection supports Maximum Fragment Length Negotiation Extension (max_fragment_length, see RFC6066) or you know for certain that it will never send a message longer than a certain number of bytes.

If the value is set too low, symptoms are a failed TLS handshake or a return value of MBEDTLS_ERR_SSL_INVALID_RECORD (-0x7200).

Range:
- from 512 to 16384

Default value:
- 16384

CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN

Asymmetric in/out fragment length

Found in: Component config > mbedTLS

If enabled, this option allows customizing TLS in/out fragment length in asymmetric way. Please note that enabling this with default values saves 12KB of dynamic memory per TLS connection.

Default value:
- Yes (enabled)

CONFIG_MBEDTLS_SSL_IN_CONTENT_LEN

TLS maximum incoming fragment length

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN

This defines maximum incoming fragment length, overriding default maximum content length (MBEDTLS_SSL_MAX_CONTENT_LEN).

Range:
- from 512 to 16384

Default value:
- 16384

CONFIG_MBEDTLS_SSL_OUT_CONTENT_LEN

TLS maximum outgoing fragment length

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN

This defines maximum outgoing fragment length, overriding default maximum content length (MBEDTLS_SSL_MAX_CONTENT_LEN).

Range:
- from 512 to 16384

Default value:
- 4096

CONFIG_MBEDTLS_DYNAMIC_BUFFER

Using dynamic TX/RX buffer

Found in: Component config > mbedTLS

Using dynamic TX/RX buffer. After enabling this option, mbedTLS will allocate TX buffer when need to send data and then free it if all data is sent, allocate RX buffer when need to receive data and then free it when all data is used or read by upper layer.

By default, when SSL is initialized, mbedTLS also allocate TX and RX buffer with the default value of "MBEDTLS_SSL_OUT_CONTENT_LEN" or "MBEDTLS_SSL_IN_CONTENT_LEN" , so to save more heap, users can set the options to be an appropriate value.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_SSL_PROTO_DTLS* && *CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH*

CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA

Free private key and DHM data after its usage

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DYNAMIC_BUFFER

Free private key and DHM data after its usage in handshake process.

The option will decrease heap cost when handshake, but also lead to problem:

Becasue all certificate, private key and DHM data are freed so users should register certificate and private key to ssl config object again.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_DYNAMIC_BUFFER*

CONFIG_MBEDTLS_DYNAMIC_FREE_CA_CERT

Free SSL CA certificate after its usage

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DYNAMIC_BUFFER > CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA

Free CA certificate after its usage in the handshake process. This option will decrease the heap footprint for the TLS handshake, but may lead to a problem: If the respective ssl object needs to perform the TLS handshake again, the CA certificate should once again be registered to the ssl object.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA*

CONFIG_MBEDTLS_DEBUG

Enable mbedTLS debugging

Found in: Component config > mbedTLS

Enable mbedTLS debugging functions at compile time.

If this option is enabled, you can include "mbedtls/esp_debug.h" and call mbedtls_esp_enable_debug_log() at runtime in order to enable mbedTLS debug output via the ESP log mechanism.

Default value:

- No (disabled)

CONFIG_MBEDTLS_DEBUG_LEVEL

Set mbedTLS debugging level

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DEBUG

Set mbedTLS debugging level

Available options:

- Warning (MBEDTLS_DEBUG_LEVEL_WARN)
- Info (MBEDTLS_DEBUG_LEVEL_INFO)
- Debug (MBEDTLS_DEBUG_LEVEL_DEBUG)

- Verbose (MBEDTLS_DEBUG_LEVEL_VERBOSE)

mbedTLS v3.x related Contains:

- *DTLS-based configurations*
- *CONFIG_MBEDTLS_PKCS7_C*
- *CONFIG_MBEDTLS_SSL_CONTEXT_SERIALIZATION*
- *CONFIG_MBEDTLS_X509_TRUSTED_CERT_CALLBACK*
- *CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE*
- *CONFIG_MBEDTLS_SSL_CID_PADDING_GRANULARITY*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*
- *CONFIG_MBEDTLS_ECDH_LEGACY_CONTEXT*
- *CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH*

CONFIG_MBEDTLS_SSL_PROTO_TLS1_3

Support TLS 1.3 protocol

Found in: Component config > mbedTLS > mbedTLS v3.x related

Default value:

- No (disabled) if *CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE* && *CONFIG_MBEDTLS_DYNAMIC_BUFFER*

TLS 1.3 related configurations Contains:

- *CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_EPHEMERAL*
- *CONFIG_MBEDTLS_SSL_TLS1_3_COMPATIBILITY_MODE*
- *CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK_EPHEMERAL*
- *CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK*

CONFIG_MBEDTLS_SSL_TLS1_3_COMPATIBILITY_MODE

TLS 1.3 middlebox compatibility mode

Found in: Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*

CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK

TLS 1.3 PSK key exchange mode

Found in: Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*

CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_EPHEMERAL

TLS 1.3 ephemeral key exchange mode

Found in: Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*

CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK_EPHEMERAL

TLS 1.3 PSK ephemeral key exchange mode

Found in: Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3*

CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH

Variable SSL buffer length

Found in: Component config > mbedTLS > mbedTLS v3.x related

This enables the SSL buffer to be resized automatically based on the negotiated maximum fragment length in each direction.

Default value:

- No (disabled)

CONFIG_MBEDTLS_ECDH_LEGACY_CONTEXT

Use a backward compatible ECDH context (Experimental)

Found in: Component config > mbedTLS > mbedTLS v3.x related

Use the legacy ECDH context format. Define this option only if you enable MBEDTLS_ECP_RESTARTABLE or if you want to access ECDH context fields directly.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_ECDH_C* && *CONFIG_MBEDTLS_ECP_RESTARTABLE*

CONFIG_MBEDTLS_X509_TRUSTED_CERT_CALLBACK

Enable trusted certificate callbacks

Found in: Component config > mbedTLS > mbedTLS v3.x related

Enables users to configure the set of trusted certificates through a callback instead of a linked list.

See mbedTLS documentation for required API and more details.

Default value:

- No (disabled)

CONFIG_MBEDTLS_SSL_CONTEXT_SERIALIZATION

Enable serialization of the TLS context structures

Found in: Component config > mbedTLS > mbedTLS v3.x related

Enable serialization of the TLS context structures This is a local optimization in handling a single, potentially long-lived connection.

See mbedTLS documentation for required API and more details. Disabling this option will save some code size.

Default value:

- No (disabled)

CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE

Keep peer certificate after handshake completion

Found in: Component config > mbedTLS > mbedTLS v3.x related

Keep the peer's certificate after completion of the handshake. Disabling this option will save about 4kB of heap and some code size.

See mbedTLS documentation for required API and more details.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_PKCS7_C

Enable PKCS #7

Found in: Component config > mbedTLS > mbedTLS v3.x related

Enable PKCS #7 core for using PKCS #7-formatted signatures.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_SSL_CID_PADDING_GRANULARITY

Record plaintext padding

Found in: Component config > mbedTLS > mbedTLS v3.x related

Controls the use of record plaintext padding in TLS 1.3 and when using the Connection ID extension in DTLS 1.2.

The padding will always be chosen so that the length of the padded plaintext is a multiple of the value of this option.

Notes: A value of 1 means that no padding will be used for outgoing records. On systems lacking division instructions, a power of two should be preferred.

Range:

- from 0 to 32 if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3* || *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID*

Default value:

- 16 if *CONFIG_MBEDTLS_SSL_PROTO_TLS1_3* || *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID*

DTLS-based configurations Contains:

- *CONFIG_MBEDTLS_SSL_DTLS_SRTP*
- *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID*

CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID

Support for the DTLS Connection ID extension

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations

Enable support for the DTLS Connection ID extension which allows to identify DTLS connections across changes in the underlying transport.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

CONFIG_MBEDTLS_SSL_CID_IN_LEN_MAX

Maximum length of CIDs used for incoming DTLS messages

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations > CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID

Maximum length of CIDs used for incoming DTLS messages

Range:

- from 0 to 32 if *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID* && *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

Default value:

- 32 if *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID* && *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

CONFIG_MBEDTLS_SSL_CID_OUT_LEN_MAX

Maximum length of CIDs used for outgoing DTLS messages

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations > CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID

Maximum length of CIDs used for outgoing DTLS messages

Range:

- from 0 to 32 if *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID* && *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

Default value:

- 32 if *CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID* && *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

CONFIG_MBEDTLS_SSL_DTLS_SRTP

Enable support for negotiation of DTLS-SRTP (RFC 5764)

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations

Enable support for negotiation of DTLS-SRTP (RFC 5764) through the use_srtp extension.

See mbedTLS documentation for required API and more details. Disabling this option will save some code size.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_SSL_PROTO_DTLS*

Certificate Bundle Contains:

- *CONFIG_MBEDTLS_CERTIFICATE_BUNDLE*

CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

Enable trusted root certificate bundle

Found in: Component config > mbedTLS > Certificate Bundle

Enable support for large number of default root certificates

When enabled this option allows user to store default as well as customer specific root certificates in compressed format rather than storing full certificate. For the root certificates the public key and the subject name will be stored.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_DEFAULT_CERTIFICATE_BUNDLE

Default certificate bundle options

Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

Available options:

- Use the full default certificate bundle (MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_FULL)
- Use only the most common certificates from the default bundles (MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_CMN)
  Use only the most common certificates from the default bundles, reducing the size with 50%, while still having around 99% coverage.
- Do not use the default certificate bundle (MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_NONE)

CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE

Add custom certificates to the default bundle

Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

Default value:

- No (disabled)

CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE_PATH

Custom certificate bundle path

Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE > CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE

Name of the custom certificate directory or file. This path is evaluated relative to the project root directory.

CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_MAX_CERTS

Maximum no of certificates allowed in certificate bundle

Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

Default value:

- 200

CONFIG_MBEDTLS_ECP_RESTARTABLE

Enable mbedTLS ecp restartable

Found in: Component config > mbedTLS

Enable "non-blocking" ECC operations that can return early and be resumed.

Default value:

- No (disabled)

CONFIG_MBEDTLS_CMAC_C

Enable CMAC mode for block ciphers

Found in: Component config > mbedTLS

Enable the CMAC (Cipher-based Message Authentication Code) mode for block ciphers.

Default value:

- No (disabled)

CONFIG_MBEDTLS_HARDWARE_AES

Enable hardware AES acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated AES encryption & decryption.

Note that if the ESP32 CPU is running at 240MHz, hardware AES does not offer any speed boost over software AES.

Default value:

- Yes (enabled) if SPIRAM_CACHE_WORKAROUND_STRATEGY_DUPLDST

CONFIG_MBEDTLS_HARDWARE_GCM

Enable partially hardware accelerated GCM

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_AES

Enable partially hardware accelerated GCM. GHASH calculation is still done in software.

If MBEDTLS_HARDWARE_GCM is disabled and MBEDTLS_HARDWARE_AES is enabled then mbedTLS will still use the hardware accelerated AES block operation, but on a single block at a time.

Default value:

- Yes (enabled) if SOC_AES_SUPPORT_GCM && *CONFIG_MBEDTLS_HARDWARE_AES*

CONFIG_MBEDTLS_GCM_SUPPORT_NON_AES_CIPHER

Enable support for non-AES ciphers in GCM operation

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_AES

Enable this config to support fallback to software definitions for a non-AES cipher GCM operation as we support hardware acceleration only for AES cipher. Some of the non-AES ciphers used in a GCM operation are DES, ARIA, CAMELLIA, CHACHA20, BLOWFISH.

If this config is disabled, performing a non-AES cipher GCM operation with the config MBEDTLS_HARDWARE_AES enabled will result in calculation of an AES-GCM operation instead for the given input values and thus could lead to failure in certificate validation which would ultimately lead to a SSL handshake failure.

This config being by-default enabled leads to an increase in binary size footprint of ~2.5KB. In case you are sure that your use case (for example, client and server configurations in case of a TLS handshake) would not involve any GCM operations using a non-AES cipher, you can safely disable this config, leading to reduction in binary size footprint.

Default value:

- No (disabled)

CONFIG_MBEDTLS_HARDWARE_MPI

Enable hardware MPI (bignum) acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated multiple precision integer operations.

Hardware accelerated multiplication, modulo multiplication, and modular exponentiation for up to SOC_RSA_MAX_BIT_LEN bit results.

These operations are used by RSA.

Default value:

- Yes (enabled) if SPIRAM_CACHE_WORKAROUND_STRATEGY_DUPLDST

CONFIG_MBEDTLS_HARDWARE_SHA

Enable hardware SHA acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated SHA1, SHA256, SHA384 & SHA512 in mbedTLS.

Due to a hardware limitation, on the ESP32 hardware acceleration is only guaranteed if SHA digests are calculated one at a time. If more than one SHA digest is calculated at the same time, one will be calculated fully in hardware and the rest will be calculated (at least partially calculated) in software. This happens automatically.

SHA hardware acceleration is faster than software in some situations but slower in others. You should benchmark to find the best setting for you.

Default value:

- Yes (enabled) if SPIRAM_CACHE_WORKAROUND_STRATEGY_DUPLDST

CONFIG_MBEDTLS_HARDWARE_ECC

Enable hardware ECC acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated ECC point multiplication and point verification for points on curve SECP192R1 and SECP256R1 in mbedTLS

Default value:

- Yes (enabled) if SOC_ECC_SUPPORTED

CONFIG_MBEDTLS_ECC_OTHER_CURVES_SOFT_FALLBACK

Fallback to software implementation for curves not supported in hardware

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_ECC

Fallback to software implementation of ECC point multiplication and point verification for curves not supported in hardware.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_HARDWARE_ECC*

CONFIG_MBEDTLS_ROM_MD5

Use MD5 implementation in ROM

Found in: Component config > mbedTLS

Use ROM MD5 in mbedTLS.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN

Enable hardware ECDSA sign acceleration when using ATECC608A

Found in: Component config > mbedTLS

This option enables hardware acceleration for ECDSA sign function, only when using ATECC608A cryptoauth chip (integrated with ESP32-WROOM-32SE)

Default value:

- No (disabled)

CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY

Enable hardware ECDSA verify acceleration when using ATECC608A

Found in: Component config > mbedTLS

This option enables hardware acceleration for ECDSA sign function, only when using ATECC608A cryptoauth chip (integrated with ESP32-WROOM-32SE)

Default value:

- No (disabled)

CONFIG_MBEDTLS_HAVE_TIME

Enable mbedtls time support

Found in: Component config > mbedTLS

Enable use of time.h functions (time() and gmtime()) by mbedTLS.

This option doesn't require the system time to be correct, but enables functionality that requires relative timekeeping - for example periodic expiry of TLS session tickets or session cache entries.

Disabling this option will save some firmware size, particularly if the rest of the firmware doesn't call any standard timekeeeping functions.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_PLATFORM_TIME_ALT

Enable mbedtls time support: platform-specific

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HAVE_TIME

Enabling this config will provide users with a function "mbedtls_platform_set_time()" that allows to set an alternative time function pointer.

Default value:

- No (disabled)

CONFIG_MBEDTLS_HAVE_TIME_DATE

Enable mbedtls certificate expiry check

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HAVE_TIME

Enables X.509 certificate expiry checks in mbedTLS.

If this option is disabled (default) then X.509 certificate "valid from" and "valid to" timestamp fields are ignored.

If this option is enabled, these fields are compared with the current system date and time. The time is retrieved using the standard time() and gmtime() functions. If the certificate is not valid for the current system time then verification will fail with code MBEDTLS_X509_BADCERT_FUTURE or MBEDTLS_X509_BADCERT_EXPIRED.

Enabling this option requires adding functionality in the firmware to set the system clock to a valid timestamp before using TLS. The recommended way to do this is via ESP-IDF's SNTP functionality, but any method can be used.

In the case where only a small number of certificates are trusted by the device, please carefully consider the tradeoffs of enabling this option. There may be undesired consequences, for example if all trusted

certificates expire while the device is offline and a TLS connection is required to update. Or if an issue with the SNTP server means that the system time is invalid for an extended period after a reset.

Default value:

- No (disabled)

CONFIG_MBEDTLS_ECDSA_DETERMINISTIC

Enable deterministic ECDSA

Found in: Component config > mbedTLS

Standard ECDSA is "fragile" in the sense that lack of entropy when signing may result in a compromise of the long-term signing key.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_SHA512_C

Enable the SHA-384 and SHA-512 cryptographic hash algorithms

Found in: Component config > mbedTLS

Enable MBEDTLS_SHA512_C adds support for SHA-384 and SHA-512.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_TLS_MODE

TLS Protocol Role

Found in: Component config > mbedTLS

mbedTLS can be compiled with protocol support for the TLS server, TLS client, or both server and client.

Reducing the number of TLS roles supported saves code size.

Available options:

- Server & Client (MBEDTLS_TLS_SERVER_AND_CLIENT)
- Server (MBEDTLS_TLS_SERVER_ONLY)
- Client (MBEDTLS_TLS_CLIENT_ONLY)
- None (MBEDTLS_TLS_DISABLED)

TLS Key Exchange Methods

Contains:

- *CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECJPAKE*
- *CONFIG_MBEDTLS_PSK_MODES*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE*

CONFIG_MBEDTLS_PSK_MODES

Enable pre-shared-key ciphersuites

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to show configuration for different types of pre-shared-key TLS authentatication methods.

Leaving this options disabled will save code size if they are not used.

Default value:

- No (disabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_PSK

Enable PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_PSK_MODES

Enable to support symmetric key PSK (pre-shared-key) TLS key exchange modes.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_PSK_MODES*

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK

Enable DHE-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_PSK_MODES

Enable to support Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_PSK_MODES* && *CONFIG_MBEDTLS_DHM_C*

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK

Enable ECDHE-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_PSK_MODES

Enable to support Elliptic-Curve-Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_PSK_MODES* && *CONFIG_MBEDTLS_ECDH_C*

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK

Enable RSA-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_PSK_MODES

Enable to support RSA PSK (pre-shared-key) TLS authentication modes.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_PSK_MODES*

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA

Enable RSA-only based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-RSA-WITH-

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA

Enable DHE-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-DHE-RSA-WITH-

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_DHM_C*

CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Support Elliptic Curve based ciphersuites

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to show Elliptic Curve based ciphersuite mode options.

Disabling all Elliptic Curve ciphersuites saves code size and can give slightly faster TLS handshakes, provided the server supports RSA-only ciphersuite modes.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA

Enable ECDHE-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA

Enable ECDHE-ECDSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA

Enable ECDH-ECDSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA

Enable ECDH-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods > CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_KEY_EXCHANGE_ECJPAKE

Enable ECJPAKE based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-ECJPAKE-WITH-

Default value:

- No (disabled) if *CONFIG_MBEDTLS_ECJPAKE_C* && *CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED*

CONFIG_MBEDTLS_SSL_RENEGOTIATION

Support TLS renegotiation

Found in: Component config > mbedTLS

The two main uses of renegotiation are (1) refresh keys on long-lived connections and (2) client authentication after the initial handshake. If you don't need renegotiation, disabling it will save code size and reduce the possibility of abuse/vulnerability.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_SSL_PROTO_TLS1_2

Support TLS 1.2 protocol

Found in: Component config > mbedTLS

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_SSL_PROTO_GMTSSL1_1

Support GM/T SSL 1.1 protocol

Found in: Component config > mbedTLS

Provisions for GM/T SSL 1.1 support

Default value:

- No (disabled)

CONFIG_MBEDTLS_SSL_PROTO_DTLS

Support DTLS protocol (all versions)

Found in: Component config > mbedTLS

Requires TLS 1.2 to be enabled for DTLS 1.2

Default value:

- No (disabled)

CONFIG_MBEDTLS_SSL_ALPN

Support ALPN (Application Layer Protocol Negotiation)

Found in: Component config > mbedTLS

Disabling this option will save some code size if it is not needed.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS

TLS: Client Support for RFC 5077 SSL session tickets

Found in: Component config > mbedTLS

Client support for RFC 5077 session tickets. See mbedTLS documentation for more details. Disabling this option will save some code size.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS

TLS: Server Support for RFC 5077 SSL session tickets

Found in: Component config > mbedTLS

Server support for RFC 5077 session tickets. See mbedTLS documentation for more details. Disabling this option will save some code size.

Default value:

- Yes (enabled)

Symmetric Ciphers

Contains:

- *CONFIG_MBEDTLS_AES_C*
- *CONFIG_MBEDTLS_BLOWFISH_C*
- *CONFIG_MBEDTLS_CAMELLIA_C*
- *CONFIG_MBEDTLS_CCM_C*
- *CONFIG_MBEDTLS_DES_C*
- *CONFIG_MBEDTLS_GCM_C*
- *CONFIG_MBEDTLS_NIST_KW_C*
- *CONFIG_MBEDTLS_XTEA_C*

CONFIG_MBEDTLS_AES_C

AES block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_CAMELLIA_C

Camellia block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

Default value:

- No (disabled)

CONFIG_MBEDTLS_DES_C

DES block cipher (legacy, insecure)

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the DES block cipher to support 3DES-based TLS ciphersuites.

3DES is vulnerable to the Sweet32 attack and should only be enabled if absolutely necessary.

Default value:

- No (disabled)

CONFIG_MBEDTLS_BLOWFISH_C

Blowfish block cipher (read help)

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the Blowfish block cipher (not used for TLS sessions.)

The Blowfish cipher is not used for mbedTLS TLS sessions but can be used for other purposes. Read up on the limitations of Blowfish (including Sweet32) before enabling.

Default value:

- No (disabled)

CONFIG_MBEDTLS_XTEA_C

XTEA block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the XTEA block cipher.

Default value:

- No (disabled)

CONFIG_MBEDTLS_CCM_C

CCM (Counter with CBC-MAC) block cipher modes

Found in: Component config > mbedTLS > Symmetric Ciphers

Enable Counter with CBC-MAC (CCM) modes for AES and/or Camellia ciphers.

Disabling this option saves some code size.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_GCM_C

GCM (Galois/Counter) block cipher modes

Found in: Component config > mbedTLS > Symmetric Ciphers

Enable Galois/Counter Mode for AES and/or Camellia ciphers.

This option is generally faster than CCM.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_NIST_KW_C

NIST key wrapping (KW) and KW padding (KWP)

Found in: Component config > mbedTLS > Symmetric Ciphers

Enable NIST key wrapping and key wrapping padding.

Default value:

- No (disabled)

CONFIG_MBEDTLS_RIPEMD160_C

Enable RIPEMD-160 hash algorithm

Found in: Component config > mbedTLS

Enable the RIPEMD-160 hash algorithm.

Default value:

- No (disabled)

Certificates Contains:

- *CONFIG_MBEDTLS_PEM_PARSE_C*
- *CONFIG_MBEDTLS_PEM_WRITE_C*
- *CONFIG_MBEDTLS_X509_CRL_PARSE_C*
- *CONFIG_MBEDTLS_X509_CSR_PARSE_C*

CONFIG_MBEDTLS_PEM_PARSE_C

Read & Parse PEM formatted certificates

Found in: Component config > mbedTLS > Certificates

Enable decoding/parsing of PEM formatted certificates.

If your certificates are all in the simpler DER format, disabling this option will save some code size.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_PEM_WRITE_C

Write PEM formatted certificates

Found in: Component config > mbedTLS > Certificates

Enable writing of PEM formatted certificates.

If writing certificate data only in DER format, disabling this option will save some code size.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_X509_CRL_PARSE_C

X.509 CRL parsing

Found in: Component config > mbedTLS > Certificates

Support for parsing X.509 Certifificate Revocation Lists.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_X509_CSR_PARSE_C

X.509 CSR parsing

Found in: Component config > mbedTLS > Certificates

Support for parsing X.509 Certifificate Signing Requests

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ECP_C

Elliptic Curve Ciphers

Found in: Component config > mbedTLS

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_DHM_C

Diffie-Hellman-Merkle key exchange (DHM)

Found in: Component config > mbedTLS

Enable DHM. Needed to use DHE-xxx TLS ciphersuites.

Note that the security of Diffie-Hellman key exchanges depends on a suitable prime being used for the exchange. Please see detailed warning text about this in file *mbedtls/dhm.h* file.

Default value:

- No (disabled)

CONFIG_MBEDTLS_ECDH_C

Elliptic Curve Diffie-Hellman (ECDH)

Found in: Component config > mbedTLS

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ECDSA_C

Elliptic Curve DSA

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_ECDH_C

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ECJPAKE_C

Elliptic curve J-PAKE

Found in: Component config > mbedTLS

Enable ECJPAKE. Needed to use ECJPAKE-xxx TLS ciphersuites.

Default value:

- No (disabled)

CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED

Enable SECP192R1 curve

Found in: Component config > mbedTLS

Enable support for SECP192R1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED

Enable SECP224R1 curve

Found in: Component config > mbedTLS

Enable support for SECP224R1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED

Enable SECP256R1 curve

Found in: Component config > mbedTLS

Enable support for SECP256R1 Elliptic Curve.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED

Enable SECP384R1 curve

Found in: Component config > mbedTLS

Enable support for SECP384R1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED

Enable SECP521R1 curve

Found in: Component config > mbedTLS

Enable support for SECP521R1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED

Enable SECP192K1 curve

Found in: Component config > mbedTLS

Enable support for SECP192K1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED

Enable SECP224K1 curve

Found in: Component config > mbedTLS

Enable support for SECP224K1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED

Enable SECP256K1 curve

Found in: Component config > mbedTLS

Enable support for SECP256K1 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED

Enable BP256R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED

Enable BP384R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED

Enable BP512R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED

Enable CURVE25519 curve

Found in: Component config > mbedTLS

Enable support for CURVE25519 Elliptic Curve.

Default value:

- Yes (enabled) if (*CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN* || *CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY*) && *CONFIG_MBEDTLS_ECP_C*

CONFIG_MBEDTLS_ECP_NIST_OPTIM

NIST ‘modulo p’ optimisations

Found in: Component config > mbedTLS

NIST ‘modulo p’ optimisations increase Elliptic Curve operation performance.

Disabling this option saves some code size.

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_ECP_FIXED_POINT_OPTIM

Enable fixed-point multiplication optimisations

Found in: Component config > mbedTLS

This configuration option enables optimizations to speedup (about 3 ~ 4 times) the ECP fixed point multiplication using pre-computed tables in the flash memory. Disabling this configuration option saves flash footprint (about 29KB if all Elliptic Curve selected) in the application binary.

end of Elliptic Curve options

Default value:

- Yes (enabled)

CONFIG_MBEDTLS_POLY1305_C

Poly1305 MAC algorithm

Found in: Component config > mbedTLS

Enable support for Poly1305 MAC algorithm.

Default value:

- No (disabled)

CONFIG_MBEDTLS_CHACHA20_C

Chacha20 stream cipher

Found in: Component config > mbedTLS

Enable support for Chacha20 stream cipher.

Default value:

- No (disabled)

CONFIG_MBEDTLS_CHACHAPOLY_C

ChaCha20-Poly1305 AEAD algorithm

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_CHACHA20_C

Enable support for ChaCha20-Poly1305 AEAD algorithm.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_CHACHA20_C* && *CONFIG_MBEDTLS_POLY1305_C*

CONFIG_MBEDTLS_HKDF_C

HKDF algorithm (RFC 5869)

Found in: Component config > mbedTLS

Enable support for the Hashed Message Authentication Code (HMAC)-based key derivation function (HKDF).

Default value:

- No (disabled)

CONFIG_MBEDTLS_THREADING_C

Enable the threading abstraction layer

Found in: Component config > mbedTLS

If you do intend to use contexts between threads, you will need to enable this layer to prevent race conditions.

Default value:

- No (disabled)

CONFIG_MBEDTLS_THREADING_ALT

Enable threading alternate implementation

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_THREADING_C

Enable threading alt to allow your own alternate threading implementation.

Default value:

- Yes (enabled) if *CONFIG_MBEDTLS_THREADING_C*

CONFIG_MBEDTLS_THREADING_PTHREAD

Enable threading pthread implementation

Found in: Component config > mbedTLS > CONFIG_MBEDTLS_THREADING_C

Enable the pthread wrapper layer for the threading layer.

Default value:

- No (disabled) if *CONFIG_MBEDTLS_THREADING_C*

CONFIG_MBEDTLS_LARGE_KEY_SOFTWARE_MPI

Fallback to software implementation for larger MPI values

Found in: Component config > mbedTLS

Fallback to software implementation for RSA key lengths larger than SOC_RSA_MAX_BIT_LEN. If this is not active then the ESP will be unable to process keys greater than SOC_RSA_MAX_BIT_LEN.

Default value:

- No (disabled)

CONFIG_MBEDTLS_USE_CRYPTO_ROM_IMPL

Use ROM implementation of the crypto algorithm

Found in: Component config > mbedTLS

Enable this flag to use mbedtls crypto algorithm from ROM instead of ESP-IDF.

This configuration option saves flash footprint in the application binary. Note that the version of mbedtls crypto algorithm library in ROM is v2.16.12. We have done the security analysis of the mbedtls revision in ROM (v2.16.12) and ensured that affected symbols have been patched (removed). If in the future mbedtls revisions there are security issues that also affects the version in ROM (v2.16.12) then we shall patch the relevant symbols. This would increase the flash footprint and hence care must be taken to keep some reserved space for the application binary in flash layout.

Default value:

- No (disabled) if ESP_ROM_HAS_MBEDTLS_CRYPTO_LIB

ESP-MQTT Configurations

Contains:

- *CONFIG_MQTT_CUSTOM_OUTBOX*
- *CONFIG_MQTT_TRANSPORT_SSL*
- *CONFIG_MQTT_TRANSPORT_WEBSOCKET*
- *CONFIG_MQTT_PROTOCOL_311*
- *CONFIG_MQTT_PROTOCOL_5*
- *CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED*
- *CONFIG_MQTT_USE_CUSTOM_CONFIG*
- *CONFIG_MQTT_OUTBOX_EXPIRED_TIMEOUT_MS*
- *CONFIG_MQTT_REPORT_DELETED_MESSAGES*
- *CONFIG_MQTT_SKIP_PUBLISH_IF_DISCONNECTED*
- *CONFIG_MQTT_MSG_ID_INCREMENTAL*

CONFIG_MQTT_PROTOCOL_311

Enable MQTT protocol 3.1.1

Found in: Component config > ESP-MQTT Configurations

If not, this library will use MQTT protocol 3.1

Default value:

- Yes (enabled)

CONFIG_MQTT_PROTOCOL_5

Enable MQTT protocol 5.0

Found in: Component config > ESP-MQTT Configurations

If not, this library will not support MQTT 5.0

Default value:

- No (disabled)

CONFIG_MQTT_TRANSPORT_SSL

Enable MQTT over SSL

Found in: Component config > ESP-MQTT Configurations

Enable MQTT transport over SSL with mbedtls

Default value:

- Yes (enabled)

CONFIG_MQTT_TRANSPORT_WEBSOCKET

Enable MQTT over Websocket

Found in: Component config > ESP-MQTT Configurations

Enable MQTT transport over Websocket.

Default value:

- Yes (enabled)

CONFIG_MQTT_TRANSPORT_WEBSOCKET_SECURE

Enable MQTT over Websocket Secure

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_TRANSPORT_WEBSOCKET

Enable MQTT transport over Websocket Secure.

Default value:

- Yes (enabled)

CONFIG_MQTT_MSG_ID_INCREMENTAL

Use Incremental Message Id

Found in: Component config > ESP-MQTT Configurations

Set this to true for the message id (2.3.1 Packet Identifier) to be generated as an incremental number rather then a random value (used by default)

Default value:

- No (disabled)

CONFIG_MQTT_SKIP_PUBLISH_IF_DISCONNECTED

Skip publish if disconnected

Found in: Component config > ESP-MQTT Configurations

Set this to true to avoid publishing (enqueueing messages) if the client is disconnected. The MQTT client tries to publish all messages by default, even in the disconnected state (where the qos1 and qos2 packets are stored in the internal outbox to be published later) The MQTT_SKIP_PUBLISH_IF_DISCONNECTED option allows applications to override this behaviour and not enqueue publish packets in the disconnected state.

Default value:

- No (disabled)

CONFIG_MQTT_REPORT_DELETED_MESSAGES

Report deleted messages

Found in: Component config > ESP-MQTT Configurations

Set this to true to post events for all messages which were deleted from the outbox before being correctly sent and confirmed.

Default value:

- No (disabled)

CONFIG_MQTT_USE_CUSTOM_CONFIG

MQTT Using custom configurations

Found in: Component config > ESP-MQTT Configurations

Custom MQTT configurations.

Default value:

- No (disabled)

CONFIG_MQTT_TCP_DEFAULT_PORT

Default MQTT over TCP port

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

Default MQTT over TCP port

Default value:

- 1883 if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

CONFIG_MQTT_SSL_DEFAULT_PORT

Default MQTT over SSL port

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

Default MQTT over SSL port

Default value:

- 8883 if *CONFIG_MQTT_USE_CUSTOM_CONFIG* && *CONFIG_MQTT_TRANSPORT_SSL*

CONFIG_MQTT_WS_DEFAULT_PORT

Default MQTT over Websocket port

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

Default MQTT over Websocket port

Default value:

- 80 if *CONFIG_MQTT_USE_CUSTOM_CONFIG* && *CONFIG_MQTT_TRANSPORT_WEBSOCKET*

CONFIG_MQTT_WSS_DEFAULT_PORT

Default MQTT over Websocket Secure port

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

Default MQTT over Websocket Secure port

Default value:

- 443 if *CONFIG_MQTT_USE_CUSTOM_CONFIG* && *CONFIG_MQTT_TRANSPORT_WEBSOCKET* && *CONFIG_MQTT_TRANSPORT_WEBSOCKET_SECURE*

CONFIG_MQTT_BUFFER_SIZE

Default MQTT Buffer Size

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

This buffer size using for both transmit and receive

Default value:

- 1024 if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

CONFIG_MQTT_TASK_STACK_SIZE

MQTT task stack size

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

MQTT task stack size

Default value:

- 6144 if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

CONFIG_MQTT_DISABLE_API_LOCKS

Disable API locks

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

Default config employs API locks to protect internal structures. It is possible to disable these locks if the user code doesn't access MQTT API from multiple concurrent tasks

Default value:

- No (disabled) if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

CONFIG_MQTT_TASK_PRIORITY

MQTT task priority

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG

MQTT task priority. Higher number denotes higher priority.

Default value:

- 5 if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED

Enable MQTT task core selection

Found in: Component config > ESP-MQTT Configurations

This will enable core selection

CONFIG_MQTT_TASK_CORE_SELECTION

Core to use ?

Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED

Available options:

- Core 0 (MQTT_USE_CORE_0)
- Core 1 (MQTT_USE_CORE_1)

CONFIG_MQTT_CUSTOM_OUTBOX

Enable custom outbox implementation

Found in: Component config > ESP-MQTT Configurations

Set to true if a specific implementation of message outbox is needed (e.g. persistent outbox in NVM or similar). Note: Implementation of the custom outbox must be added to the mqtt component. These CMake commands could be used to append the custom implementation to lib-mqtt sources: idf_component_get_property(mqtt mqtt COMPONENT_LIB) set_property(TARGET ${mqtt} PROPERTY SOURCES ${PROJECT_DIR}/custom_outbox.c APPEND)

Default value:

- No (disabled)

CONFIG_MQTT_OUTBOX_EXPIRED_TIMEOUT_MS

Outbox message expired timeout[ms]

Found in: Component config > ESP-MQTT Configurations

Messages which stays in the outbox longer than this value before being published will be discarded.

Default value:

- 30000 if *CONFIG_MQTT_USE_CUSTOM_CONFIG*

Newlib Contains:

- *CONFIG_NEWLIB_NANO_FORMAT*
- *CONFIG_NEWLIB_STDIN_LINE_ENDING*
- *CONFIG_NEWLIB_STDOUT_LINE_ENDING*
- *CONFIG_NEWLIB_TIME_SYSCALL*

CONFIG_NEWLIB_STDOUT_LINE_ENDING

Line ending for UART output

Found in: Component config > Newlib

This option allows configuring the desired line endings sent to UART when a newline ('n', LF) appears on stdout. Three options are possible:

CRLF: whenever LF is encountered, prepend it with CR

LF: no modification is applied, stdout is sent as is

CR: each occurence of LF is replaced with CR

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Available options:

- CRLF (NEWLIB_STDOUT_LINE_ENDING_CRLF)
- LF (NEWLIB_STDOUT_LINE_ENDING_LF)
- CR (NEWLIB_STDOUT_LINE_ENDING_CR)

CONFIG_NEWLIB_STDIN_LINE_ENDING

Line ending for UART input

Found in: Component config > Newlib

This option allows configuring which input sequence on UART produces a newline ('n', LF) on stdin. Three options are possible:

CRLF: CRLF is converted to LF

LF: no modification is applied, input is sent to stdin as is

CR: each occurence of CR is replaced with LF

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Available options:

- CRLF (NEWLIB_STDIN_LINE_ENDING_CRLF)
- LF (NEWLIB_STDIN_LINE_ENDING_LF)
- CR (NEWLIB_STDIN_LINE_ENDING_CR)

CONFIG_NEWLIB_NANO_FORMAT

Enable 'nano' formatting options for printf/scanf family

Found in: Component config > Newlib

ESP32 ROM contains parts of newlib C library, including printf/scanf family of functions. These functions have been compiled with so-called "nano" formatting option. This option doesn't support 64-bit integer formats and C99 features, such as positional arguments.

For more details about "nano" formatting option, please see newlib readme file, search for '–enable-newlib-nano-formatted-io': https://sourceware.org/newlib/README

If this option is enabled, build system will use functions available in ROM, reducing the application binary size. Functions available in ROM run faster than functions which run from flash. Functions available in ROM can also run when flash instruction cache is disabled.

If you need 64-bit integer formatting support or C99 features, keep this option disabled.

CONFIG_NEWLIB_TIME_SYSCALL

Timers used for gettimeofday function

Found in: Component config > Newlib

This setting defines which hardware timers are used to implement 'gettimeofday' and 'time' functions in C library.

- **If both high-resolution (systimer for all targets except ESP32)** and RTC timers are used, timekeeping will continue in deep sleep. Time will be reported at 1 microsecond resolution. This is the default, and the recommended option.
- **If only high-resolution timer (systimer) is used, gettimeofday will** provide time at microsecond resolution. Time will not be preserved when going into deep sleep mode.
- **If only RTC timer is used, timekeeping will continue in** deep sleep, but time will be measured at 6.(6) microsecond resolution. Also the gettimeofday function itself may take longer to run.
- **If no timers are used, gettimeofday and time functions** return -1 and set errno to ENOSYS.
- **When RTC is used for timekeeping, two RTC_STORE registers are** used to keep time in deep sleep mode.

Available options:

- RTC and high-resolution timer (NEWLIB_TIME_SYSCALL_USE_RTC_HRT)
- RTC (NEWLIB_TIME_SYSCALL_USE_RTC)
- High-resolution timer (NEWLIB_TIME_SYSCALL_USE_HRT)
- None (NEWLIB_TIME_SYSCALL_USE_NONE)

NVS Contains:

- *CONFIG_NVS_ENCRYPTION*
- *CONFIG_NVS_COMPATIBLE_PRE_V4_3_ENCRYPTION_FLAG*
- *CONFIG_NVS_ASSERT_ERROR_CHECK*

CONFIG_NVS_ENCRYPTION

Enable NVS encryption

Found in: Component config > NVS

This option enables encryption for NVS. When enabled, AES-XTS is used to encrypt the complete NVS data, except the page headers. It requires XTS encryption keys to be stored in an encrypted partition. This means enabling flash encryption is a pre-requisite for this feature.

Default value:

- Yes (enabled) if *CONFIG_SECURE_FLASH_ENC_ENABLED*

CONFIG_NVS_COMPATIBLE_PRE_V4_3_ENCRYPTION_FLAG

NVS partition encrypted flag compatible with ESP-IDF before v4.3

Found in: Component config > NVS

Enabling this will ignore "encrypted" flag for NVS partitions. NVS encryption scheme is different than hardware flash encryption and hence it is not recommended to have "encrypted" flag for NVS partitions. This was not being checked in pre v4.3 IDF. Hence, if you have any devices where this flag is kept enabled in partition table then enabling this config will allow to have same behavior as pre v4.3 IDF.

CONFIG_NVS_ASSERT_ERROR_CHECK

Use assertions for error checking

Found in: Component config > NVS

This option switches error checking type between assertions (y) or return codes (n).

Default value:

- No (disabled)

OpenThread Contains:

- *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_ENABLED

OpenThread

Found in: Component config > OpenThread

Select this option to enable OpenThread and show the submenu with OpenThread configuration choices.

Default value:

- No (disabled)

CONFIG_OPENTHREAD_LOG_LEVEL_DYNAMIC

Enable dynamic log level control

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable dynamic log level control for OpenThread

Default value:

- Yes (enabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_LOG_LEVEL

OpenThread log verbosity

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select OpenThread log level.

Available options:

- No logs (OPENTHREAD_LOG_LEVEL_NONE)
- Error logs (OPENTHREAD_LOG_LEVEL_CRIT)
- Warning logs (OPENTHREAD_LOG_LEVEL_WARN)
- Notice logs (OPENTHREAD_LOG_LEVEL_NOTE)
- Info logs (OPENTHREAD_LOG_LEVEL_INFO)
- Debug logs (OPENTHREAD_LOG_LEVEL_DEBG)

CONFIG_OPENTHREAD_RADIO_TYPE

Config the Thread radio type

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Configure how OpenThread connects to the 15.4 radio

Available options:

- Native 15.4 radio (OPENTHREAD_RADIO_NATIVE)
  Select this to use the native 15.4 radio.
- Connect via UART (OPENTHREAD_RADIO_SPINEL_UART)
  Select this to connect to a Radio Co-Processor via UART.

CONFIG_OPENTHREAD_DEVICE_TYPE

Config the Thread device type

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

OpenThread can be configured to different device types (FTD, MTD, Radio)

Available options:

- Full Thread Device (OPENTHREAD_FTD)
  Select this to enable Full Thread Device which can act as router and leader in a Thread network.
- Minimal Thread Device (OPENTHREAD_MTD)
  Select this to enable Minimal Thread Device which can only act as end device in a Thread network. This will reduce the code size of the OpenThread stack.
- Radio Only Device (OPENTHREAD_RADIO)
  Select this to enable Radio Only Device which can only forward 15.4 packets to the host. The OpenThread stack will be run on the host and OpenThread will have minimal footprint on the radio only device.

CONFIG_OPENTHREAD_CLI

Enable Openthread Command-Line Interface

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable Command-Line Interface in OpenThread.

Default value:

- Yes (enabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_DIAG

Enable diag

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable Diag in OpenThread. This will enable diag mode and a series of diag commands in the OpenThread command line. These commands allow users to manipulate low-level features of the storage and 15.4 radio.

Default value:

- Yes (enabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_COMMISSIONER

Enable Commissioner

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable commissioner in OpenThread. This will enable the device to act as a commissioner in the Thread network. A commissioner checks the pre-shared key from a joining device with the Thread commissioning protocol and shares the network parameter with the joining device upon success.

Default value:

- No (disabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_JOINER

Enable Joiner

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable Joiner in OpenThread. This allows a device to join the Thread network with a pre-shared key using the Thread commissioning protocol.

Default value:

- No (disabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_SRP_CLIENT

Enable SRP Client

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable SRP Client in OpenThread. This allows a device to register SRP services to SRP Server.

Default value:

- No (disabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_BORDER_ROUTER

Enable Border Router

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to enable border router features in OpenThread.

Default value:
- No (disabled) if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_NUM_MESSAGE_BUFFERS

The number of openthread message buffers

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Range:
- from 50 to 100 if *CONFIG_OPENTHREAD_ENABLED*

Default value:
- 65 if *CONFIG_OPENTHREAD_ENABLED*

CONFIG_OPENTHREAD_DNS64_CLIENT

Use dns64 client

Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED

Select this option to acquire NAT64 address from dns servers.

Default value:
- No (disabled) if *CONFIG_OPENTHREAD_ENABLED*

Protocomm Contains:

- *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0*
- *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1*
- *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2*

CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0

Support protocomm security version 0 (no security)

Found in: Component config > Protocomm

Enable support of security version 0. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

Default value:
- Yes (enabled)

CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1

Support protocomm security version 1 (Curve25519 key exchange + AES-CTR encryption/decryption)

Found in: Component config > Protocomm

Enable support of security version 1. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

Default value:
- Yes (enabled)

CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2

Support protocomm security version 2 (SRP6a-based key exchange + AES-GCM encryption/decryption)

Found in: Component config > Protocomm

Enable support of security version 2. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

Default value:

- Yes (enabled)

PThreads Contains:

- *CONFIG_PTHREAD_TASK_NAME_DEFAULT*
- *CONFIG_PTHREAD_TASK_CORE_DEFAULT*
- *CONFIG_PTHREAD_TASK_PRIO_DEFAULT*
- *CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT*
- *CONFIG_PTHREAD_STACK_MIN*

CONFIG_PTHREAD_TASK_PRIO_DEFAULT

Default task priority

Found in: Component config > PThreads

Priority used to create new tasks with default pthread parameters.

Range:

- from 0 to 255

Default value:

- 5

CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT

Default task stack size

Found in: Component config > PThreads

Stack size used to create new tasks with default pthread parameters.

Default value:

- 3072

CONFIG_PTHREAD_STACK_MIN

Minimum allowed pthread stack size

Found in: Component config > PThreads

Minimum allowed pthread stack size set in attributes passed to pthread_create

Default value:

- 768

CONFIG_PTHREAD_TASK_CORE_DEFAULT

Default pthread core affinity

Found in: Component config > PThreads

The default core to which pthreads are pinned.

Available options:

- No affinity (PTHREAD_DEFAULT_CORE_NO_AFFINITY)
- Core 0 (PTHREAD_DEFAULT_CORE_0)
- Core 1 (PTHREAD_DEFAULT_CORE_1)

CONFIG_PTHREAD_TASK_NAME_DEFAULT

Default name of pthreads

Found in: Component config > PThreads

The default name of pthreads.

Default value:

- "pthread"

SoC Settings Contains:

- *MMU Config*

MMU Config

Main Flash configuration Contains:

- *Optional and Experimental Features (READ DOCS FIRST)*
- *SPI Flash behavior when brownout*

SPI Flash behavior when brownout Contains:

- *CONFIG_SPI_FLASH_BROWNOUT_RESET_XMC*

CONFIG_SPI_FLASH_BROWNOUT_RESET_XMC

Enable sending reset when brownout for XMC flash chips

Found in: Component config > Main Flash configuration > SPI Flash behavior when brownout

When this option is selected, the patch will be enabled for XMC. Follow the recommended flow by XMC for better stability.

DO NOT DISABLE UNLESS YOU KNOW WHAT YOU ARE DOING.

Default value:

- Yes (enabled) if APP_BUILD_TYPE_PURE_RAM_APP

Optional and Experimental Features (READ DOCS FIRST) Contains:

- *CONFIG_SPI_FLASH_HPM_DC*

CONFIG_SPI_FLASH_HPM_DC

Support HPM using DC (READ DOCS FIRST)

Found in: Component config > Main Flash configuration > Optional and Experimental Features (READ DOCS FIRST)

This feature needs your bootloader to be compiled DC-aware (BOOTLOADER_FLASH_DC_AWARE=y). Otherwise the chip will not be able to boot after a reset.

Available options:

- Auto (Enable when bootloader support enabled (BOOTLOADER_FLASH_DC_AWARE)) (SPI_FLASH_HPM_DC_AUTO)
- Disable (READ DOCS FIRST) (SPI_FLASH_HPM_DC_DISABLE)

SPI Flash driver Contains:

- *Auto-detect flash chips*
- *CONFIG_SPI_FLASH_BYPASS_BLOCK_ERASE*
- *CONFIG_SPI_FLASH_ENABLE_ENCRYPTED_READ_WRITE*
- *CONFIG_SPI_FLASH_ENABLE_COUNTERS*
- *CONFIG_SPI_FLASH_ROM_DRIVER_PATCH*
- *CONFIG_SPI_FLASH_YIELD_DURING_ERASE*
- *CONFIG_SPI_FLASH_CHECK_ERASE_TIMEOUT_DISABLED*
- *CONFIG_SPI_FLASH_WRITE_CHUNK_SIZE*
- *CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST*
- *CONFIG_SPI_FLASH_SIZE_OVERRIDE*
- *CONFIG_SPI_FLASH_SHARE_SPI1_BUS*
- *CONFIG_SPI_FLASH_VERIFY_WRITE*
- *CONFIG_SPI_FLASH_DANGEROUS_WRITE*

CONFIG_SPI_FLASH_VERIFY_WRITE

Verify SPI flash writes

Found in: Component config > SPI Flash driver

If this option is enabled, any time SPI flash is written then the data will be read back and verified. This can catch hardware problems with SPI flash, or flash which was not erased before verification.

Default value:

- No (disabled)

CONFIG_SPI_FLASH_LOG_FAILED_WRITE

Log errors if verification fails

Found in: Component config > SPI Flash driver > CONFIG_SPI_FLASH_VERIFY_WRITE

If this option is enabled, if SPI flash write verification fails then a log error line will be written with the address, expected & actual values. This can be useful when debugging hardware SPI flash problems.

Default value:

- No (disabled) if *CONFIG_SPI_FLASH_VERIFY_WRITE*

CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE

Log warning if writing zero bits to ones

Found in: Component config > SPI Flash driver > CONFIG_SPI_FLASH_VERIFY_WRITE

If this option is enabled, any SPI flash write which tries to set zero bits in the flash to ones will log a warning. Such writes will not result in the requested data appearing identically in flash once written, as SPI NOR flash can only set bits to one when an entire sector is erased. After erasing, individual bits can only be written from one to zero.

Note that some software (such as SPIFFS) which is aware of SPI NOR flash may write one bits as an optimisation, relying on the data in flash becoming a bitwise AND of the new data and any existing data. Such software will log spurious warnings if this option is enabled.

Default value:

- No (disabled) if *CONFIG_SPI_FLASH_VERIFY_WRITE*

CONFIG_SPI_FLASH_ENABLE_COUNTERS

Enable operation counters

Found in: Component config > SPI Flash driver

This option enables the following APIs:

- spi_flash_reset_counters
- spi_flash_dump_counters
- spi_flash_get_counters

These APIs may be used to collect performance data for spi_flash APIs and to help understand behaviour of libraries which use SPI flash.

Default value:

- 0

CONFIG_SPI_FLASH_ROM_DRIVER_PATCH

Enable SPI flash ROM driver patched functions

Found in: Component config > SPI Flash driver

Enable this flag to use patched versions of SPI flash ROM driver functions. This option should be enabled, if any one of the following is true: (1) need to write to flash on ESP32-D2WD; (2) main SPI flash is connected to non-default pins; (3) main SPI flash chip is manufactured by ISSI.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_DANGEROUS_WRITE

Writing to dangerous flash regions

Found in: Component config > SPI Flash driver

SPI flash APIs can optionally abort or return a failure code if erasing or writing addresses that fall at the beginning of flash (covering the bootloader and partition table) or that overlap the app partition that contains the running app.

It is not recommended to ever write to these regions from an IDF app, and this check prevents logic errors or corrupted firmware memory from damaging these regions.

Note that this feature *does not* check calls to the esp_rom_xxx SPI flash ROM functions. These functions should not be called directly from IDF applications.

Available options:

- Aborts (SPI_FLASH_DANGEROUS_WRITE_ABORTS)
- Fails (SPI_FLASH_DANGEROUS_WRITE_FAILS)
- Allowed (SPI_FLASH_DANGEROUS_WRITE_ALLOWED)

CONFIG_SPI_FLASH_SHARE_SPI1_BUS

Support other devices attached to SPI1 bus

Found in: Component config > SPI Flash driver

Each SPI bus needs a lock for arbitration among devices. This allows multiple devices on a same bus, but may reduce the speed of esp_flash driver access to the main flash chip.

If you only need to use esp_flash driver to access the main flash chip, disable this option, and the lock will be bypassed on SPI1 bus. Otherwise if extra devices are needed to attach to SPI1 bus, enable this option.

Default value:

- No (disabled)

CONFIG_SPI_FLASH_BYPASS_BLOCK_ERASE

Bypass a block erase and always do sector erase

Found in: Component config > SPI Flash driver

Some flash chips can have very high "max" erase times, especially for block erase (32KB or 64KB). This option allows to bypass "block erase" and always do sector erase commands. This will be much slower overall in most cases, but improves latency for other code to run.

Default value:

- No (disabled)

CONFIG_SPI_FLASH_YIELD_DURING_ERASE

Enables yield operation during flash erase

Found in: Component config > SPI Flash driver

This allows to yield the CPUs between erase commands. Prevents starvation of other tasks. Please use this configuration together with `SPI\_FLASH\_ERASE\_YIELD\_DURATION\_MS` and `SPI\_FLASH\_ERASE\_YIELD\_TICKS` after carefully checking flash datasheet to avoid a watchdog timeout. For more information, please check *SPI Flash API* reference documenation under section *OS Function*.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS

Duration of erasing to yield CPUs (ms)

Found in: Component config > SPI Flash driver > CONFIG_SPI_FLASH_YIELD_DURING_ERASE

If a duration of one erase command is large then it will yield CPUs after finishing a current command.

Default value:

- 20

CONFIG_SPI_FLASH_ERASE_YIELD_TICKS

CPU release time (tick) for an erase operation

Found in: Component config > SPI Flash driver > CONFIG_SPI_FLASH_YIELD_DURING_ERASE

Defines how many ticks will be before returning to continue a erasing.

Default value:

- 1

CONFIG_SPI_FLASH_WRITE_CHUNK_SIZE

Flash write chunk size

Found in: Component config > SPI Flash driver

Flash write is broken down in terms of multiple (smaller) write operations. This configuration options helps to set individual write chunk size, smaller value here ensures that cache (and non-IRAM resident interrupts) remains disabled for shorter duration.

Range:

- from 256 to 8192

Default value:

- 8192

CONFIG_SPI_FLASH_SIZE_OVERRIDE

Override flash size in bootloader header by ESPTOOLPY_FLASHSIZE

Found in: Component config > SPI Flash driver

SPI Flash driver uses the flash size configured in bootloader header by default. Enable this option to override flash size with latest ESPTOOLPY_FLASHSIZE value from the app header if the size in the bootloader header is incorrect.

Default value:

- No (disabled)

CONFIG_SPI_FLASH_CHECK_ERASE_TIMEOUT_DISABLED

Flash timeout checkout disabled

Found in: Component config > SPI Flash driver

This option is helpful if you are using a flash chip whose timeout is quite large or unpredictable.

Default value:

- No (disabled)

CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST

Override default chip driver list

Found in: Component config > SPI Flash driver

This option allows the chip driver list to be customized, instead of using the default list provided by ESP-IDF.

When this option is enabled, the default list is no longer compiled or linked. Instead, the *default_registered_chips* structure must be provided by the user.

See example: custom_chip_driver under examples/storage for more details.

Default value:

- No (disabled)

Auto-detect flash chips Contains:

- *CONFIG_SPI_FLASH_SUPPORT_BOYA_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_GD_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_ISSI_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_MXIC_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_TH_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_WINBOND_CHIP*

CONFIG_SPI_FLASH_SUPPORT_ISSI_CHIP

ISSI

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of ISSI chips if chip vendor not directly given by `chip\_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_SUPPORT_MXIC_CHIP

MXIC

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of MXIC chips if chip vendor not directly given by `chip\_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_SUPPORT_GD_CHIP

GigaDevice

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of GD (GigaDevice) chips if chip vendor not directly given by `chip\_drv` member of the chip struct. If you are using Wrover modules, please don't disable this, otherwise your flash may not work in 4-bit mode.

This adds support for variant chips, however will extend detecting time and image size. Note that the default chip driver supports the GD chips with product ID 60H.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_SUPPORT_WINBOND_CHIP

Winbond

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of Winbond chips if chip vendor not directly given by `chip\_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

Default value:

- Yes (enabled)

CONFIG_SPI_FLASH_SUPPORT_BOYA_CHIP

BOYA

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of BOYA chips if chip vendor not directly given by `chip\_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

Default value:

- No (disabled)
- Yes (enabled)

CONFIG_SPI_FLASH_SUPPORT_TH_CHIP

TH

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of TH chips if chip vendor not directly given by `chip\_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

Default value:

- No (disabled)
- Yes (enabled)

CONFIG_SPI_FLASH_ENABLE_ENCRYPTED_READ_WRITE

Enable encrypted partition read/write operations

Found in: Component config > SPI Flash driver

This option enables flash read/write operations to encrypted partition/s. This option is kept enabled irrespective of state of flash encryption feature. However, in case application is not using flash encryption feature and is in need of some additional memory from IRAM region (~1KB) then this config can be disabled.

Default value:

- Yes (enabled)

SPIFFS Configuration Contains:

- *Debug Configuration*
- *CONFIG_SPIFFS_USE_MAGIC*
- *CONFIG_SPIFFS_GC_STATS*
- *CONFIG_SPIFFS_PAGE_CHECK*
- *CONFIG_SPIFFS_FOLLOW_SYMLINKS*
- *CONFIG_SPIFFS_MAX_PARTITIONS*
- *CONFIG_SPIFFS_USE_MTIME*
- *CONFIG_SPIFFS_GC_MAX_RUNS*
- *CONFIG_SPIFFS_OBJ_NAME_LEN*
- *CONFIG_SPIFFS_META_LENGTH*
- *SPIFFS Cache Configuration*
- *CONFIG_SPIFFS_PAGE_SIZE*
- *CONFIG_SPIFFS_MTIME_WIDE_64_BITS*

CONFIG_SPIFFS_MAX_PARTITIONS

Maximum Number of Partitions

Found in: Component config > SPIFFS Configuration

Define maximum number of partitions that can be mounted.

Range:

- from 1 to 10

Default value:

- 3

SPIFFS Cache Configuration Contains:

- *CONFIG_SPIFFS_CACHE*

CONFIG_SPIFFS_CACHE

Enable SPIFFS Cache

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration

Enables/disable memory read caching of nucleus file system operations.

Default value:

- Yes (enabled)

CONFIG_SPIFFS_CACHE_WR

Enable SPIFFS Write Caching

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration > CONFIG_SPIFFS_CACHE

Enables memory write caching for file descriptors in hydrogen.

Default value:

- Yes (enabled)

CONFIG_SPIFFS_CACHE_STATS

Enable SPIFFS Cache Statistics

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration > CONFIG_SPIFFS_CACHE

Enable/disable statistics on caching. Debug/test purpose only.

Default value:

- No (disabled)

CONFIG_SPIFFS_PAGE_CHECK

Enable SPIFFS Page Check

Found in: Component config > SPIFFS Configuration

Always check header of each accessed page to ensure consistent state. If enabled it will increase number of reads from flash, especially if cache is disabled.

Default value:

- Yes (enabled)

CONFIG_SPIFFS_GC_MAX_RUNS

Set Maximum GC Runs

Found in: Component config > SPIFFS Configuration

Define maximum number of GC runs to perform to reach desired free pages.

Range:

- from 1 to 10000

Default value:

- 10

CONFIG_SPIFFS_GC_STATS

Enable SPIFFS GC Statistics

Found in: Component config > SPIFFS Configuration

Enable/disable statistics on gc. Debug/test purpose only.

Default value:

- No (disabled)

CONFIG_SPIFFS_PAGE_SIZE

SPIFFS logical page size

Found in: Component config > SPIFFS Configuration

Logical page size of SPIFFS partition, in bytes. Must be multiple of flash page size (which is usually 256 bytes). Larger page sizes reduce overhead when storing large files, and improve filesystem performance when reading large files. Smaller page sizes reduce overhead when storing small (< page size) files.

Range:
- from 256 to 1024

Default value:
- 256

CONFIG_SPIFFS_OBJ_NAME_LEN

Set SPIFFS Maximum Name Length

Found in: Component config > SPIFFS Configuration

Object name maximum length. Note that this length include the zero-termination character, meaning maximum string of characters can at most be SPIFFS_OBJ_NAME_LEN - 1.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

Range:
- from 1 to 256

Default value:
- 32

CONFIG_SPIFFS_FOLLOW_SYMLINKS

Enable symbolic links for image creation

Found in: Component config > SPIFFS Configuration

If this option is enabled, symbolic links are taken into account during partition image creation.

Default value:
- No (disabled)

CONFIG_SPIFFS_USE_MAGIC

Enable SPIFFS Filesystem Magic

Found in: Component config > SPIFFS Configuration

Enable this to have an identifiable spiffs filesystem. This will look for a magic in all sectors to determine if this is a valid spiffs system or not at mount time.

Default value:
- Yes (enabled)

CONFIG_SPIFFS_USE_MAGIC_LENGTH

Enable SPIFFS Filesystem Length Magic

Found in: Component config > SPIFFS Configuration > CONFIG_SPIFFS_USE_MAGIC

If this option is enabled, the magic will also be dependent on the length of the filesystem. For example, a filesystem configured and formatted for 4 megabytes will not be accepted for mounting with a configuration defining the filesystem as 2 megabytes.

Default value:

- Yes (enabled)

CONFIG_SPIFFS_META_LENGTH

Size of per-file metadata field

Found in: Component config > SPIFFS Configuration

This option sets the number of extra bytes stored in the file header. These bytes can be used in an application-specific manner. Set this to at least 4 bytes to enable support for saving file modification time.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

Default value:

- 4

CONFIG_SPIFFS_USE_MTIME

Save file modification time

Found in: Component config > SPIFFS Configuration

If enabled, then the first 4 bytes of per-file metadata will be used to store file modification time (mtime), accessible through stat/fstat functions. Modification time is updated when the file is opened.

Default value:

- Yes (enabled)

CONFIG_SPIFFS_MTIME_WIDE_64_BITS

The time field occupies 64 bits in the image instead of 32 bits

Found in: Component config > SPIFFS Configuration

If this option is not set, the time field is 32 bits (up to 2106 year), otherwise it is 64 bits and make sure it matches SPIFFS_META_LENGTH. If the chip already has the spiffs image with the time field = 32 bits then this option cannot be applied in this case. Erase it first before using this option. To resolve the Y2K38 problem for the spiffs, use a toolchain with 64-bit time_t support.

Default value:

- No (disabled) if *CONFIG_SPIFFS_META_LENGTH* >= 8

Debug Configuration

Contains:

- *CONFIG_SPIFFS_DBG*
- *CONFIG_SPIFFS_API_DBG*
- *CONFIG_SPIFFS_CACHE_DBG*
- *CONFIG_SPIFFS_CHECK_DBG*
- *CONFIG_SPIFFS_TEST_VISUALISATION*
- *CONFIG_SPIFFS_GC_DBG*

CONFIG_SPIFFS_DBG

Enable general SPIFFS debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print general debug mesages to the console.

Default value:

- No (disabled)

CONFIG_SPIFFS_API_DBG

Enable SPIFFS API debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print API debug mesages to the console.

Default value:

- No (disabled)

CONFIG_SPIFFS_GC_DBG

Enable SPIFFS Garbage Cleaner debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print GC debug mesages to the console.

Default value:

- No (disabled)

CONFIG_SPIFFS_CACHE_DBG

Enable SPIFFS Cache debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print cache debug mesages to the console.

Default value:

- No (disabled)

CONFIG_SPIFFS_CHECK_DBG

Enable SPIFFS Filesystem Check debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print Filesystem Check debug mesages to the console.

Default value:

- No (disabled)

CONFIG_SPIFFS_TEST_VISUALISATION

Enable SPIFFS Filesystem Visualization

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enable this option to enable SPIFFS_vis function in the API.

Default value:

- No (disabled)

TCP Transport Contains:

- *Websocket*

Websocket Contains:

- *CONFIG_WS_TRANSPORT*

CONFIG_WS_TRANSPORT

Enable Websocket Transport

Found in: Component config > TCP Transport > Websocket

Enable support for creating websocket transport.

Default value:
- Yes (enabled)

CONFIG_WS_BUFFER_SIZE

Websocket transport buffer size

Found in: Component config > TCP Transport > Websocket > CONFIG_WS_TRANSPORT

Size of the buffer used for constructing the HTTP Upgrade request during connect

Default value:
- 1024

CONFIG_WS_DYNAMIC_BUFFER

Using dynamic websocket transport buffer

Found in: Component config > TCP Transport > Websocket > CONFIG_WS_TRANSPORT

If enable this option, websocket transport buffer will be freed after connection succeed to save more heap.

Default value:
- No (disabled)

Ultra Low Power (ULP) Co-processor

Contains:

- *CONFIG_ULP_COPROC_ENABLED*
- *ULP RISC-V Settings*

CONFIG_ULP_COPROC_ENABLED

Enable Ultra Low Power (ULP) Co-processor

Found in: Component config > Ultra Low Power (ULP) Co-processor

Enable this feature if you plan to use the ULP Co-processor. Once this option is enabled, further ULP co-processor configuration will appear in the menu.

Default value:
- No (disabled)

CONFIG_ULP_COPROC_TYPE

ULP Co-processor type

Found in: Component config > Ultra Low Power (ULP) Co-processor > CONFIG_ULP_COPROC_ENABLED

Choose the ULP Coprocessor type: ULP FSM (Finite State Machine) or ULP RISC-V. Please note that ESP32 only supports ULP FSM.

Available options:
- ULP FSM (Finite State Machine) (ULP_COPROC_TYPE_FSM)
- ULP RISC-V (ULP_COPROC_TYPE_RISCV)

CONFIG_ULP_COPROC_RESERVE_MEM

RTC slow memory reserved for coprocessor

Found in: Component config > Ultra Low Power (ULP) Co-processor > CONFIG_ULP_COPROC_ENABLED

Bytes of memory to reserve for ULP Co-processor firmware & data. Data is reserved at the beginning of RTC slow memory.

Range:
- from 32 to 8176 if *CONFIG_ULP_COPROC_ENABLED*

Default value:
- 512 if *CONFIG_ULP_COPROC_ENABLED*

ULP RISC-V Settings

Unity unit testing library

Contains:

- *CONFIG_UNITY_ENABLE_COLOR*
- *CONFIG_UNITY_ENABLE_IDF_TEST_RUNNER*
- *CONFIG_UNITY_ENABLE_FIXTURE*
- *CONFIG_UNITY_ENABLE_BACKTRACE_ON_FAIL*
- *CONFIG_UNITY_ENABLE_64BIT*
- *CONFIG_UNITY_ENABLE_DOUBLE*
- *CONFIG_UNITY_ENABLE_FLOAT*

CONFIG_UNITY_ENABLE_FLOAT

Support for float type

Found in: Component config > Unity unit testing library

If not set, assertions on float arguments will not be available.

Default value:
- Yes (enabled)

CONFIG_UNITY_ENABLE_DOUBLE

Support for double type

Found in: Component config > Unity unit testing library

If not set, assertions on double arguments will not be available.

Default value:
- Yes (enabled)

CONFIG_UNITY_ENABLE_64BIT

Support for 64-bit integer types

Found in: Component config > Unity unit testing library

If not set, assertions on 64-bit integer types will always fail. If this feature is enabled, take care not to pass pointers (which are 32 bit) to UNITY_ASSERT_EQUAL, as that will cause pointer-to-int-cast warnings.

Default value:
- No (disabled)

CONFIG_UNITY_ENABLE_COLOR

Colorize test output

Found in: Component config > Unity unit testing library

If set, Unity will colorize test results using console escape sequences.

Default value:

- No (disabled)

CONFIG_UNITY_ENABLE_IDF_TEST_RUNNER

Include ESP-IDF test registration/running helpers

Found in: Component config > Unity unit testing library

If set, then the following features will be available:

- TEST_CASE macro which performs automatic registration of test functions
- Functions to run registered test functions: unity_run_all_tests, unity_run_tests_with_filter, unity_run_single_test_by_name.
- Interactive menu which lists test cases and allows choosing the tests to be run, available via unity_run_menu function.

Disable if a different test registration mechanism is used.

Default value:

- Yes (enabled)

CONFIG_UNITY_ENABLE_FIXTURE

Include Unity test fixture

Found in: Component config > Unity unit testing library

If set, unity_fixture.h header file and associated source files are part of the build. These provide an optional set of macros and functions to implement test groups.

Default value:

- No (disabled)

CONFIG_UNITY_ENABLE_BACKTRACE_ON_FAIL

Print a backtrace when a unit test fails

Found in: Component config > Unity unit testing library

If set, the unity framework will print the backtrace information before jumping back to the test menu. The jumping is usually occurs in assert functions such as TEST_ASSERT, TEST_FAIL etc.

Default value:

- No (disabled)

Virtual file system

Contains:

- *CONFIG_VFS_SUPPORT_IO*

CONFIG_VFS_SUPPORT_IO

Provide basic I/O functions

Found in: Component config > Virtual file system

If enabled, the following functions are provided by the VFS component.

open, close, read, write, pread, pwrite, lseek, fstat, fsync, ioctl, fcntl

Filesystem drivers can then be registered to handle these functions for specific paths.

Disabling this option can save memory when the support for these functions is not required.

Note that the following functions can still be used with socket file descriptors when this option is disabled:

close, read, write, ioctl, fcntl.

Default value:

- Yes (enabled)

CONFIG_VFS_SUPPORT_DIR

Provide directory related functions

Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO

If enabled, the following functions are provided by the VFS component.

stat, link, unlink, rename, utime, access, truncate, rmdir, mkdir, opendir, closedir, readdir, readdir_r, seekdir, telldir, rewinddir

Filesystem drivers can then be registered to handle these functions for specific paths.

Disabling this option can save memory when the support for these functions is not required.

Default value:

- Yes (enabled)

CONFIG_VFS_SUPPORT_SELECT

Provide select function

Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO

If enabled, select function is provided by the VFS component, and can be used on peripheral file descriptors (such as UART) and sockets at the same time.

If disabled, the default select implementation will be provided by LWIP for sockets only.

Disabling this option can reduce code size if support for "select" on UART file descriptors is not required.

Default value:

- Yes (enabled) if *CONFIG_VFS_SUPPORT_IO* && *CONFIG_LWIP_USE_ONLY_LWIP_SELECT*

CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT

Suppress select() related debug outputs

Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO > CONFIG_VFS_SUPPORT_SELECT

Select() related functions might produce an unconveniently lot of debug outputs when one sets the default log level to DEBUG or higher. It is possible to suppress these debug outputs by enabling this option.

Default value:

- Yes (enabled)

CONFIG_VFS_SUPPORT_TERMIOS

Provide termios.h functions

Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO

Disabling this option can save memory when the support for termios.h is not required.

Default value:

- Yes (enabled)

Host File System I/O (Semihosting) Contains:

- *CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS*

CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS

Host FS: Maximum number of the host filesystem mount points

Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO > Host File System I/O (Semihosting)

Define maximum number of host filesystem mount points.

Default value:

- 1

Wear Levelling Contains:

- *CONFIG_WL_SECTOR_MODE*
- *CONFIG_WL_SECTOR_SIZE*

CONFIG_WL_SECTOR_SIZE

Wear Levelling library sector size

Found in: Component config > Wear Levelling

Sector size used by wear levelling library. You can set default sector size or size that will fit to the flash device sector size.

With sector size set to 4096 bytes, wear levelling library is more efficient. However if FAT filesystem is used on top of wear levelling library, it will need more temporary storage: 4096 bytes for each mounted filesystem and 4096 bytes for each opened file.

With sector size set to 512 bytes, wear levelling library will perform more operations with flash memory, but less RAM will be used by FAT filesystem library (512 bytes for the filesystem and 512 bytes for each file opened).

Available options:

- 512 (WL_SECTOR_SIZE_512)
- 4096 (WL_SECTOR_SIZE_4096)

CONFIG_WL_SECTOR_MODE

Sector store mode

Found in: Component config > Wear Levelling

Specify the mode to store data into flash:

- In Performance mode a data will be stored to the RAM and then stored back to the flash. Compared to the Safety mode, this operation is faster, but if power will be lost when erase sector operation is in progress, then the data from complete flash device sector will be lost.

- In Safety mode data from complete flash device sector will be read from flash, modified, and then stored back to flash. Compared to the Performance mode, this operation is slower, but if power is lost during erase sector operation, then the data from full flash device sector will not be lost.

Available options:
- Perfomance (WL_SECTOR_MODE_PERF)
- Safety (WL_SECTOR_MODE_SAFE)

Wi-Fi Provisioning Manager Contains:

- *CONFIG_WIFI_PROV_BLE_BONDING*
- *CONFIG_WIFI_PROV_BLE_SEC_CONN*
- *CONFIG_WIFI_PROV_BLE_FORCE_ENCRYPTION*
- *CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV*
- *CONFIG_WIFI_PROV_SCAN_MAX_ENTRIES*
- *CONFIG_WIFI_PROV_AUTOSTOP_TIMEOUT*
- *CONFIG_WIFI_PROV_STA_SCAN_METHOD*

CONFIG_WIFI_PROV_SCAN_MAX_ENTRIES

Max Wi-Fi Scan Result Entries

Found in: Component config > Wi-Fi Provisioning Manager

This sets the maximum number of entries of Wi-Fi scan results that will be kept by the provisioning manager

Range:
- from 1 to 255

Default value:
- 16

CONFIG_WIFI_PROV_AUTOSTOP_TIMEOUT

Provisioning auto-stop timeout

Found in: Component config > Wi-Fi Provisioning Manager

Time (in seconds) after which the Wi-Fi provisioning manager will auto-stop after connecting to a Wi-Fi network successfully.

Range:
- from 5 to 600

Default value:
- 30

CONFIG_WIFI_PROV_BLE_BONDING

Enable BLE bonding

Found in: Component config > Wi-Fi Provisioning Manager

This option is applicable only when provisioning transport is BLE.

CONFIG_WIFI_PROV_BLE_SEC_CONN

Enable BLE Secure connection flag

Found in: Component config > Wi-Fi Provisioning Manager

Used to enable Secure connection support when provisioning transport is BLE.

Default value:

- Yes (enabled) if BT_NIMBLE_ENABLED

CONFIG_WIFI_PROV_BLE_FORCE_ENCRYPTION

Force Link Encryption during characteristic Read / Write

Found in: Component config > Wi-Fi Provisioning Manager

Used to enforce link encryption when attempting to read / write characteristic

CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV

Keep BT on after provisioning is done

Found in: Component config > Wi-Fi Provisioning Manager

CONFIG_WIFI_PROV_DISCONNECT_AFTER_PROV

Terminate connection after provisioning is done

Found in: Component config > Wi-Fi Provisioning Manager > CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV

Default value:

- Yes (enabled) if *CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV*

CONFIG_WIFI_PROV_STA_SCAN_METHOD

Wifi Provisioning Scan Method

Found in: Component config > Wi-Fi Provisioning Manager

Available options:

- All Channel Scan (WIFI_PROV_STA_ALL_CHANNEL_SCAN)
  Scan will end after scanning the entire channel. This option is useful in Mesh WiFi Systems.
- Fast Scan (WIFI_PROV_STA_FAST_SCAN)
  Scan will end after an AP matching with the SSID has been detected.

Supplicant

Contains:

- *CONFIG_WPA_TESTING_OPTIONS*
- *CONFIG_WPA_WPS_SOFTAP_REGISTRAR*
- *CONFIG_WPA_11KV_SUPPORT*
- *CONFIG_WPA_11R_SUPPORT*
- *CONFIG_WPA_DPP_SUPPORT*
- *CONFIG_WPA_MBO_SUPPORT*
- *CONFIG_WPA_SUITE_B_192*
- *CONFIG_WPA_WAPI_PSK*
- *CONFIG_WPA_DEBUG_PRINT*
- *CONFIG_WPA_WPS_STRICT*
- *CONFIG_WPA_MBEDTLS_CRYPTO*

CONFIG_WPA_MBEDTLS_CRYPTO

Use MbedTLS crypto APIs

Found in: Component config > Supplicant

Select this option to enable the use of MbedTLS crypto APIs. The internal crypto support within the supplicant is limited and may not suffice for all new security features, including WPA3.

It is recommended to always keep this option enabled. Additionally, note that MbedTLS can leverage hardware acceleration if available, resulting in significantly faster cryptographic operations.

Default value:

- Yes (enabled)

CONFIG_WPA_MBEDTLS_TLS_CLIENT

Use MbedTLS TLS client for WiFi Enterprise connection

Found in: Component config > Supplicant > CONFIG_WPA_MBEDTLS_CRYPTO

Select this option to use MbedTLS TLS client for WPA2 enterprise connection. Please note that from MbedTLS-3.0 onwards, MbedTLS does not support SSL-3.0 TLS-v1.0, TLS-v1.1 versions. Incase your server is using one of these version, it is advisable to update your server. Please disable this option for compatibilty with older TLS versions.

Default value:

- Yes (enabled)

CONFIG_WPA_WAPI_PSK

Enable WAPI PSK support

Found in: Component config > Supplicant

Select this option to enable WAPI-PSK which is a Chinese National Standard Encryption for Wireless LANs (GB 15629.11-2003).

Default value:

- No (disabled)

CONFIG_WPA_SUITE_B_192

Enable NSA suite B support with 192 bit key

Found in: Component config > Supplicant

Select this option to enable 192 bit NSA suite-B. This is necessary to support WPA3 192 bit security.

CONFIG_WPA_DEBUG_PRINT

Print debug messages from WPA Supplicant

Found in: Component config > Supplicant

Select this option to print logging information from WPA supplicant, this includes handshake information and key hex dumps depending on the project logging level.

Enabling this could increase the build size ~60kb depending on the project logging level.

Default value:

- No (disabled)

CONFIG_WPA_TESTING_OPTIONS

Add DPP testing code

Found in: Component config > Supplicant

Select this to enable unity test for DPP.

Default value:

- No (disabled)

CONFIG_WPA_WPS_STRICT

Strictly validate all WPS attributes

Found in: Component config > Supplicant

Select this option to enable validate each WPS attribute rigorously. Disabling this add the workaorunds with various APs. Enabling this may cause inter operability issues with some APs.

Default value:

- No (disabled)

CONFIG_WPA_11KV_SUPPORT

Enable 802.11k, 802.11v APIs Support

Found in: Component config > Supplicant

Select this option to enable 802.11k 802.11v APIs(RRM and BTM support). Only APIs which are helpful for network assisted roaming are supported for now. Enable this option with BTM and RRM enabled in sta config to make device ready for network assisted roaming. BTM: BSS transition management enables an AP to request a station to transition to a specific AP, or to indicate to a station a set of preferred APs. RRM: Radio measurements enable STAs to understand the radio environment, it enables STAs to observe and gather data on radio link performance and on the radio environment. Current implementation adds beacon report, link measurement, neighbor report.

Default value:

- No (disabled)

CONFIG_WPA_SCAN_CACHE

Keep scan results in cache

Found in: Component config > Supplicant > CONFIG_WPA_11KV_SUPPORT

Keep scan results in cache, if not enabled, those will be flushed immediately.

Default value:

- No (disabled) if *CONFIG_WPA_11KV_SUPPORT*

CONFIG_WPA_MBO_SUPPORT

Enable Multi Band Operation Certification Support

Found in: Component config > Supplicant

Select this option to enable WiFi Multiband operation certification support.

Default value:

- No (disabled)

CONFIG_WPA_DPP_SUPPORT

Enable DPP support

Found in: Component config > Supplicant

Select this option to enable WiFi Easy Connect Support.

Default value:

- No (disabled)

CONFIG_WPA_11R_SUPPORT

Enable 802.11R (Fast Transition) Support

Found in: Component config > Supplicant

Select this option to enable WiFi Fast Transition Support.

Default value:

- No (disabled)

CONFIG_WPA_WPS_SOFTAP_REGISTRAR

Add WPS Registrar support in SoftAP mode

Found in: Component config > Supplicant

Select this option to enable WPS registrar support in softAP mode.

Default value:

- No (disabled)

Deprecated options and their replacements

- CONFIG_A2DP_ENABLE (*CONFIG_BT_A2DP_ENABLE*)
- **CONFIG_A2D_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_A2D_TRACE_LEVEL*)**
  - CONFIG_A2D_TRACE_LEVEL_NONE
  - CONFIG_A2D_TRACE_LEVEL_ERROR
  - CONFIG_A2D_TRACE_LEVEL_WARNING
  - CONFIG_A2D_TRACE_LEVEL_API
  - CONFIG_A2D_TRACE_LEVEL_EVENT
  - CONFIG_A2D_TRACE_LEVEL_DEBUG
  - CONFIG_A2D_TRACE_LEVEL_VERBOSE
- CONFIG_ADC2_DISABLE_DAC (*CONFIG_ADC_DISABLE_DAC*)
- **CONFIG_APPL_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_APPL_TRACE_LEVEL*)**
  - CONFIG_APPL_TRACE_LEVEL_NONE
  - CONFIG_APPL_TRACE_LEVEL_ERROR
  - CONFIG_APPL_TRACE_LEVEL_WARNING
  - CONFIG_APPL_TRACE_LEVEL_API
  - CONFIG_APPL_TRACE_LEVEL_EVENT
  - CONFIG_APPL_TRACE_LEVEL_DEBUG
  - CONFIG_APPL_TRACE_LEVEL_VERBOSE
- CONFIG_APP_ANTI_ROLLBACK (*CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK*)
- CONFIG_APP_ROLLBACK_ENABLE (*CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE*)
- CONFIG_APP_SECURE_VERSION (*CONFIG_BOOTLOADER_APP_SECURE_VERSION*)
- CONFIG_APP_SECURE_VERSION_SIZE_EFUSE_FIELD (*CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIEL*
- **CONFIG_AVCT_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_AVCT_TRACE_LEVEL*)**
  - CONFIG_AVCT_TRACE_LEVEL_NONE
  - CONFIG_AVCT_TRACE_LEVEL_ERROR
  - CONFIG_AVCT_TRACE_LEVEL_WARNING
  - CONFIG_AVCT_TRACE_LEVEL_API
  - CONFIG_AVCT_TRACE_LEVEL_EVENT
  - CONFIG_AVCT_TRACE_LEVEL_DEBUG
  - CONFIG_AVCT_TRACE_LEVEL_VERBOSE
- **CONFIG_AVDT_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_AVDT_TRACE_LEVEL*)**
  - CONFIG_AVDT_TRACE_LEVEL_NONE
  - CONFIG_AVDT_TRACE_LEVEL_ERROR
  - CONFIG_AVDT_TRACE_LEVEL_WARNING
  - CONFIG_AVDT_TRACE_LEVEL_API
  - CONFIG_AVDT_TRACE_LEVEL_EVENT
  - CONFIG_AVDT_TRACE_LEVEL_DEBUG

  - CONFIG_AVDT_TRACE_LEVEL_VERBOSE
- **CONFIG_AVRC_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_AVRC_TRACE_LEVEL***)
  - CONFIG_AVRC_TRACE_LEVEL_NONE
  - CONFIG_AVRC_TRACE_LEVEL_ERROR
  - CONFIG_AVRC_TRACE_LEVEL_WARNING
  - CONFIG_AVRC_TRACE_LEVEL_API
  - CONFIG_AVRC_TRACE_LEVEL_EVENT
  - CONFIG_AVRC_TRACE_LEVEL_DEBUG
  - CONFIG_AVRC_TRACE_LEVEL_VERBOSE
- CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY (*CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN*)
- CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD (*CONFIG_BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD*)
- CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM (*CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM*)
- CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED (*CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP*)
- CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT (*CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT*)
- CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK (*CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK*)
- CONFIG_BLE_MESH_GATT_PROXY (*CONFIG_BLE_MESH_GATT_PROXY_SERVER*)
- CONFIG_BLE_MESH_SCAN_DUPLICATE_EN (*CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN*)
- CONFIG_BLE_SCAN_DUPLICATE (*CONFIG_BTDM_BLE_SCAN_DUPL*)
- CONFIG_BLE_SMP_ENABLE (*CONFIG_BT_BLE_SMP_ENABLE*)
- CONFIG_BLUEDROID_MEM_DEBUG (*CONFIG_BT_BLUEDROID_MEM_DEBUG*)
- **CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE** (***CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE***)
  - CONFIG_BLUEDROID_PINNED_TO_CORE_0
  - CONFIG_BLUEDROID_PINNED_TO_CORE_1
- **CONFIG_BLUFI_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_BLUFI_TRACE_LEVEL***)
  - CONFIG_BLUFI_TRACE_LEVEL_NONE
  - CONFIG_BLUFI_TRACE_LEVEL_ERROR
  - CONFIG_BLUFI_TRACE_LEVEL_WARNING
  - CONFIG_BLUFI_TRACE_LEVEL_API
  - CONFIG_BLUFI_TRACE_LEVEL_EVENT
  - CONFIG_BLUFI_TRACE_LEVEL_DEBUG
  - CONFIG_BLUFI_TRACE_LEVEL_VERBOSE
- CONFIG_BNEP_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_BNEP_TRACE_LEVEL*)
- CONFIG_BROWNOUT_DET (*CONFIG_ESP_BROWNOUT_DET*)
- **CONFIG_BROWNOUT_DET_LVL_SEL** (***CONFIG_ESP_BROWNOUT_DET_LVL_SEL***)
  - CONFIG_BROWNOUT_DET_LVL_SEL_0
  - CONFIG_BROWNOUT_DET_LVL_SEL_1
  - CONFIG_BROWNOUT_DET_LVL_SEL_2
  - CONFIG_BROWNOUT_DET_LVL_SEL_3
  - CONFIG_BROWNOUT_DET_LVL_SEL_4
  - CONFIG_BROWNOUT_DET_LVL_SEL_5
  - CONFIG_BROWNOUT_DET_LVL_SEL_6
  - CONFIG_BROWNOUT_DET_LVL_SEL_7
- **CONFIG_BTC_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_BTC_TRACE_LEVEL***)
  - CONFIG_BTC_TRACE_LEVEL_NONE
  - CONFIG_BTC_TRACE_LEVEL_ERROR
  - CONFIG_BTC_TRACE_LEVEL_WARNING
  - CONFIG_BTC_TRACE_LEVEL_API
  - CONFIG_BTC_TRACE_LEVEL_EVENT
  - CONFIG_BTC_TRACE_LEVEL_DEBUG
  - CONFIG_BTC_TRACE_LEVEL_VERBOSE
- CONFIG_BTC_TASK_STACK_SIZE (*CONFIG_BT_BTC_TASK_STACK_SIZE*)
- CONFIG_BTDM_CONTROLLER_BLE_MAX_CONN (*CONFIG_BTDM_CTRL_BLE_MAX_CONN*)
- CONFIG_BTDM_CONTROLLER_BR_EDR_MAX_ACL_CONN (*CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN*)

- CONFIG_BTDM_CONTROLLER_BR_EDR_MAX_SYNC_CONN (*CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN*)
- CONFIG_BTDM_CONTROLLER_FULL_SCAN_SUPPORTED (*CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED*)
- **CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE** (***CONFIG_BTDM_CTRL_HCI_MODE_CHOICE***)
  - CONFIG_BTDM_CONTROLLER_HCI_MODE_VHCI
  - CONFIG_BTDM_CONTROLLER_HCI_MODE_UART_H4
- **CONFIG_BTDM_CONTROLLER_MODE** (***CONFIG_BTDM_CTRL_MODE***)
  - CONFIG_BTDM_CONTROLLER_MODE_BLE_ONLY
  - CONFIG_BTDM_CONTROLLER_MODE_BR_EDR_ONLY
  - CONFIG_BTDM_CONTROLLER_MODE_BTDM
- CONFIG_BTDM_CONTROLLER_MODEM_SLEEP (*CONFIG_BTDM_CTRL_MODEM_SLEEP*)
- CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE (*CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE*)
- **CONFIG_BTH_LOG_SDP_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_SDP_TRACE_LEVEL***)
  - CONFIG_SDP_TRACE_LEVEL_NONE
  - CONFIG_SDP_TRACE_LEVEL_ERROR
  - CONFIG_SDP_TRACE_LEVEL_WARNING
  - CONFIG_SDP_TRACE_LEVEL_API
  - CONFIG_SDP_TRACE_LEVEL_EVENT
  - CONFIG_SDP_TRACE_LEVEL_DEBUG
  - CONFIG_SDP_TRACE_LEVEL_VERBOSE
- **CONFIG_BTIF_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_BTIF_TRACE_LEVEL***)
  - CONFIG_BTIF_TRACE_LEVEL_NONE
  - CONFIG_BTIF_TRACE_LEVEL_ERROR
  - CONFIG_BTIF_TRACE_LEVEL_WARNING
  - CONFIG_BTIF_TRACE_LEVEL_API
  - CONFIG_BTIF_TRACE_LEVEL_EVENT
  - CONFIG_BTIF_TRACE_LEVEL_DEBUG
  - CONFIG_BTIF_TRACE_LEVEL_VERBOSE
- **CONFIG_BTM_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_BTM_TRACE_LEVEL***)
  - CONFIG_BTM_TRACE_LEVEL_NONE
  - CONFIG_BTM_TRACE_LEVEL_ERROR
  - CONFIG_BTM_TRACE_LEVEL_WARNING
  - CONFIG_BTM_TRACE_LEVEL_API
  - CONFIG_BTM_TRACE_LEVEL_EVENT
  - CONFIG_BTM_TRACE_LEVEL_DEBUG
  - CONFIG_BTM_TRACE_LEVEL_VERBOSE
- CONFIG_BTU_TASK_STACK_SIZE (*CONFIG_BT_BTU_TASK_STACK_SIZE*)
- CONFIG_BT_NIMBLE_MSYS1_BLOCK_COUNT (*CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT*)
- CONFIG_BT_NIMBLE_TASK_STACK_SIZE (*CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE*)
- CONFIG_CLASSIC_BT_ENABLED (*CONFIG_BT_CLASSIC_ENABLED*)
- **CONFIG_CONSOLE_UART** (***CONFIG_ESP_CONSOLE_UART***)
  - CONFIG_CONSOLE_UART_DEFAULT
  - CONFIG_CONSOLE_UART_CUSTOM
  - CONFIG_CONSOLE_UART_NONE, CONFIG_ESP_CONSOLE_UART_NONE
- CONFIG_CONSOLE_UART_BAUDRATE (*CONFIG_ESP_CONSOLE_UART_BAUDRATE*)
- **CONFIG_CONSOLE_UART_NUM** (***CONFIG_ESP_CONSOLE_UART_NUM***)
  - CONFIG_CONSOLE_UART_CUSTOM_NUM_0
  - CONFIG_CONSOLE_UART_CUSTOM_NUM_1
- CONFIG_CONSOLE_UART_RX_GPIO (*CONFIG_ESP_CONSOLE_UART_RX_GPIO*)
- CONFIG_CONSOLE_UART_TX_GPIO (*CONFIG_ESP_CONSOLE_UART_TX_GPIO*)
- CONFIG_CXX_EXCEPTIONS (*CONFIG_COMPILER_CXX_EXCEPTIONS*)
- CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE (*CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE*)
- CONFIG_DUPLICATE_SCAN_CACHE_SIZE (*CONFIG_BTDM_SCAN_DUPL_CACHE_SIZE*)
- CONFIG_EFUSE_SECURE_VERSION_EMULATE (*CONFIG_BOOTLOADER_EFUSE_SECURE_VERSION_EMULATE*)
- CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK (*CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP*)

- CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO (*CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO*)
- CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX (*CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX*)
- CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH (*CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH*)
- CONFIG_ESP32_COMPATIBLE_PRE_V2_1_BOOTLOADERS (*CONFIG_APP_COMPATIBLE_PRE_V2_1_BOOTLOADERS*)
- CONFIG_ESP32_COMPATIBLE_PRE_V3_1_BOOTLOADERS (*CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS*)
- **CONFIG_ESP32_CORE_DUMP_DECODE (*CONFIG_ESP_COREDUMP_DECODE*)**
  - CONFIG_ESP32_CORE_DUMP_DECODE_INFO
  - CONFIG_ESP32_CORE_DUMP_DECODE_DISABLE
- CONFIG_ESP32_CORE_DUMP_MAX_TASKS_NUM (*CONFIG_ESP_COREDUMP_MAX_TASKS_NUM*)
- CONFIG_ESP32_CORE_DUMP_STACK_SIZE (*CONFIG_ESP_COREDUMP_STACK_SIZE*)
- CONFIG_ESP32_CORE_DUMP_UART_DELAY (*CONFIG_ESP_COREDUMP_UART_DELAY*)
- CONFIG_ESP32_DEBUG_STUBS_ENABLE (*CONFIG_ESP_DEBUG_STUBS_ENABLE*)
- CONFIG_ESP32_GCOV_ENABLE (*CONFIG_APPTRACE_GCOV_ENABLE*)
- CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE (*CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE*)
- CONFIG_ESP32_PHY_DEFAULT_INIT_IF_INVALID (*CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID*)
- CONFIG_ESP32_PHY_INIT_DATA_ERROR (*CONFIG_ESP_PHY_INIT_DATA_ERROR*)
- CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION (*CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*)
- CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER (*CONFIG_ESP_PHY_MAX_WIFI_TX_POWER*)
- CONFIG_ESP32_PTHREAD_STACK_MIN (*CONFIG_PTHREAD_STACK_MIN*)
- **CONFIG_ESP32_PTHREAD_TASK_CORE_DEFAULT (*CONFIG_PTHREAD_TASK_CORE_DEFAULT*)**
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_NO_AFFINITY
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_0
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_1
- CONFIG_ESP32_PTHREAD_TASK_NAME_DEFAULT (*CONFIG_PTHREAD_TASK_NAME_DEFAULT*)
- CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT (*CONFIG_PTHREAD_TASK_PRIO_DEFAULT*)
- CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT (*CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT*)
- CONFIG_ESP32_REDUCE_PHY_TX_POWER (*CONFIG_ESP_PHY_REDUCE_TX_POWER*)
- CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES (*CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES*)
- CONFIG_ESP32_SUPPORT_MULTIPLE_PHY_INIT_DATA_BIN (*CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN*)
- CONFIG_ESP_GRATUITOUS_ARP (*CONFIG_LWIP_ESP_GRATUITOUS_ARP*)
- CONFIG_ESP_SYSTEM_PD_FLASH (*CONFIG_ESP_SLEEP_POWER_DOWN_FLASH*)
- CONFIG_ESP_TASK_WDT (*CONFIG_ESP_TASK_WDT_INIT*)
- CONFIG_EVENT_LOOP_PROFILING (*CONFIG_ESP_EVENT_LOOP_PROFILING*)
- CONFIG_FLASH_ENCRYPTION_ENABLED (*CONFIG_SECURE_FLASH_ENC_ENABLED*)
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE (*CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE*)
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT (*CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC*)
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT (*CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC*)
- **CONFIG_GAP_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_GAP_TRACE_LEVEL*)**
  - CONFIG_GAP_TRACE_LEVEL_NONE
  - CONFIG_GAP_TRACE_LEVEL_ERROR
  - CONFIG_GAP_TRACE_LEVEL_WARNING
  - CONFIG_GAP_TRACE_LEVEL_API
  - CONFIG_GAP_TRACE_LEVEL_EVENT
  - CONFIG_GAP_TRACE_LEVEL_DEBUG
  - CONFIG_GAP_TRACE_LEVEL_VERBOSE
- CONFIG_GARP_TMR_INTERVAL (*CONFIG_LWIP_GARP_TMR_INTERVAL*)
- CONFIG_GATTC_CACHE_NVS_FLASH (*CONFIG_BT_GATTC_CACHE_NVS_FLASH*)
- CONFIG_GATTC_ENABLE (*CONFIG_BT_GATTC_ENABLE*)

- CONFIG_GATTS_ENABLE (*CONFIG_BT_GATTS_ENABLE*)
- **CONFIG_GATTS_SEND_SERVICE_CHANGE_MODE** (***CONFIG_BT_GATTS_SEND_SERVICE_CHANGE_MODE***)
  - CONFIG_GATTS_SEND_SERVICE_CHANGE_MANUAL
  - CONFIG_GATTS_SEND_SERVICE_CHANGE_AUTO
- **CONFIG_GATT_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_GATT_TRACE_LEVEL***)
  - CONFIG_GATT_TRACE_LEVEL_NONE
  - CONFIG_GATT_TRACE_LEVEL_ERROR
  - CONFIG_GATT_TRACE_LEVEL_WARNING
  - CONFIG_GATT_TRACE_LEVEL_API
  - CONFIG_GATT_TRACE_LEVEL_EVENT
  - CONFIG_GATT_TRACE_LEVEL_DEBUG
  - CONFIG_GATT_TRACE_LEVEL_VERBOSE
- CONFIG_GDBSTUB_MAX_TASKS (*CONFIG_ESP_GDBSTUB_MAX_TASKS*)
- CONFIG_GDBSTUB_SUPPORT_TASKS (*CONFIG_ESP_GDBSTUB_SUPPORT_TASKS*)
- **CONFIG_HCI_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_HCI_TRACE_LEVEL***)
  - CONFIG_HCI_TRACE_LEVEL_NONE
  - CONFIG_HCI_TRACE_LEVEL_ERROR
  - CONFIG_HCI_TRACE_LEVEL_WARNING
  - CONFIG_HCI_TRACE_LEVEL_API
  - CONFIG_HCI_TRACE_LEVEL_EVENT
  - CONFIG_HCI_TRACE_LEVEL_DEBUG
  - CONFIG_HCI_TRACE_LEVEL_VERBOSE
- CONFIG_HFP_AG_ENABLE (*CONFIG_BT_HFP_AG_ENABLE*)
- **CONFIG_HFP_AUDIO_DATA_PATH** (***CONFIG_BT_HFP_AUDIO_DATA_PATH***)
  - CONFIG_HFP_AUDIO_DATA_PATH_PCM
  - CONFIG_HFP_AUDIO_DATA_PATH_HCI
- CONFIG_HFP_CLIENT_ENABLE (*CONFIG_BT_HFP_CLIENT_ENABLE*)
- CONFIG_HFP_ENABLE (*CONFIG_BT_HFP_ENABLE*)
- **CONFIG_HID_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_HID_TRACE_LEVEL***)
  - CONFIG_HID_TRACE_LEVEL_NONE
  - CONFIG_HID_TRACE_LEVEL_ERROR
  - CONFIG_HID_TRACE_LEVEL_WARNING
  - CONFIG_HID_TRACE_LEVEL_API
  - CONFIG_HID_TRACE_LEVEL_EVENT
  - CONFIG_HID_TRACE_LEVEL_DEBUG
  - CONFIG_HID_TRACE_LEVEL_VERBOSE
- CONFIG_INT_WDT (*CONFIG_ESP_INT_WDT*)
- CONFIG_INT_WDT_CHECK_CPU1 (*CONFIG_ESP_INT_WDT_CHECK_CPU1*)
- CONFIG_INT_WDT_TIMEOUT_MS (*CONFIG_ESP_INT_WDT_TIMEOUT_MS*)
- CONFIG_IPC_TASK_STACK_SIZE (*CONFIG_ESP_IPC_TASK_STACK_SIZE*)
- **CONFIG_L2CAP_INITIAL_TRACE_LEVEL** (***CONFIG_BT_LOG_L2CAP_TRACE_LEVEL***)
  - CONFIG_L2CAP_TRACE_LEVEL_NONE
  - CONFIG_L2CAP_TRACE_LEVEL_ERROR
  - CONFIG_L2CAP_TRACE_LEVEL_WARNING
  - CONFIG_L2CAP_TRACE_LEVEL_API
  - CONFIG_L2CAP_TRACE_LEVEL_EVENT
  - CONFIG_L2CAP_TRACE_LEVEL_DEBUG
  - CONFIG_L2CAP_TRACE_LEVEL_VERBOSE
- CONFIG_L2_TO_L3_COPY (*CONFIG_LWIP_L2_TO_L3_COPY*)
- **CONFIG_LOG_BOOTLOADER_LEVEL** (***CONFIG_BOOTLOADER_LOG_LEVEL***)
  - CONFIG_LOG_BOOTLOADER_LEVEL_NONE
  - CONFIG_LOG_BOOTLOADER_LEVEL_ERROR
  - CONFIG_LOG_BOOTLOADER_LEVEL_WARN
  - CONFIG_LOG_BOOTLOADER_LEVEL_INFO
  - CONFIG_LOG_BOOTLOADER_LEVEL_DEBUG
  - CONFIG_LOG_BOOTLOADER_LEVEL_VERBOSE
- CONFIG_MAIN_TASK_STACK_SIZE (*CONFIG_ESP_MAIN_TASK_STACK_SIZE*)

- **CONFIG_MCA_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_MCA_TRACE_LEVEL*)**
  - CONFIG_MCA_TRACE_LEVEL_NONE
  - CONFIG_MCA_TRACE_LEVEL_ERROR
  - CONFIG_MCA_TRACE_LEVEL_WARNING
  - CONFIG_MCA_TRACE_LEVEL_API
  - CONFIG_MCA_TRACE_LEVEL_EVENT
  - CONFIG_MCA_TRACE_LEVEL_DEBUG
  - CONFIG_MCA_TRACE_LEVEL_VERBOSE
- CONFIG_MCPWM_ISR_IN_IRAM (*CONFIG_MCPWM_ISR_IRAM_SAFE*)
- CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE (*CONFIG_BTDM_MESH_DUPL_SCAN_CACHE_SIZE*)
- CONFIG_NIMBLE_ACL_BUF_COUNT (*CONFIG_BT_NIMBLE_ACL_BUF_COUNT*)
- CONFIG_NIMBLE_ACL_BUF_SIZE (*CONFIG_BT_NIMBLE_ACL_BUF_SIZE*)
- CONFIG_NIMBLE_ATT_PREFERRED_MTU (*CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU*)
- CONFIG_NIMBLE_CRYPTO_STACK_MBEDTLS (*CONFIG_BT_NIMBLE_CRYPTO_STACK_MBEDTLS*)
- CONFIG_NIMBLE_DEBUG (*CONFIG_BT_NIMBLE_DEBUG*)
- CONFIG_NIMBLE_GAP_DEVICE_NAME_MAX_LEN (*CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN*)
- CONFIG_NIMBLE_HCI_EVT_BUF_SIZE (*CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE*)
- CONFIG_NIMBLE_HCI_EVT_HI_BUF_COUNT (*CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT*)
- CONFIG_NIMBLE_HCI_EVT_LO_BUF_COUNT (*CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT*)
- CONFIG_NIMBLE_HS_FLOW_CTRL (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_ITVL (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_ITVL*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_THRESH (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_THRESH*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT*)
- CONFIG_NIMBLE_L2CAP_COC_MAX_NUM (*CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM*)
- CONFIG_NIMBLE_MAX_BONDS (*CONFIG_BT_NIMBLE_MAX_BONDS*)
- CONFIG_NIMBLE_MAX_CCCDS (*CONFIG_BT_NIMBLE_MAX_CCCDS*)
- CONFIG_NIMBLE_MAX_CONNECTIONS (*CONFIG_BT_NIMBLE_MAX_CONNECTIONS*)
- **CONFIG_NIMBLE_MEM_ALLOC_MODE (*CONFIG_BT_NIMBLE_MEM_ALLOC_MODE*)**
  - CONFIG_NIMBLE_MEM_ALLOC_MODE_INTERNAL
  - CONFIG_NIMBLE_MEM_ALLOC_MODE_EXTERNAL
  - CONFIG_NIMBLE_MEM_ALLOC_MODE_DEFAULT
- CONFIG_NIMBLE_MESH (*CONFIG_BT_NIMBLE_MESH*)
- CONFIG_NIMBLE_MESH_DEVICE_NAME (*CONFIG_BT_NIMBLE_MESH_DEVICE_NAME*)
- CONFIG_NIMBLE_MESH_FRIEND (*CONFIG_BT_NIMBLE_MESH_FRIEND*)
- CONFIG_NIMBLE_MESH_GATT_PROXY (*CONFIG_BT_NIMBLE_MESH_GATT_PROXY*)
- CONFIG_NIMBLE_MESH_LOW_POWER (*CONFIG_BT_NIMBLE_MESH_LOW_POWER*)
- CONFIG_NIMBLE_MESH_PB_ADV (*CONFIG_BT_NIMBLE_MESH_PB_ADV*)
- CONFIG_NIMBLE_MESH_PB_GATT (*CONFIG_BT_NIMBLE_MESH_PB_GATT*)
- CONFIG_NIMBLE_MESH_PROV (*CONFIG_BT_NIMBLE_MESH_PROV*)
- CONFIG_NIMBLE_MESH_PROXY (*CONFIG_BT_NIMBLE_MESH_PROXY*)
- CONFIG_NIMBLE_MESH_RELAY (*CONFIG_BT_NIMBLE_MESH_RELAY*)
- CONFIG_NIMBLE_NVS_PERSIST (*CONFIG_BT_NIMBLE_NVS_PERSIST*)
- **CONFIG_NIMBLE_PINNED_TO_CORE_CHOICE (*CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE*)**
  - CONFIG_NIMBLE_PINNED_TO_CORE_0
  - CONFIG_NIMBLE_PINNED_TO_CORE_1
- CONFIG_NIMBLE_ROLE_BROADCASTER (*CONFIG_BT_NIMBLE_ROLE_BROADCASTER*)
- CONFIG_NIMBLE_ROLE_CENTRAL (*CONFIG_BT_NIMBLE_ROLE_CENTRAL*)
- CONFIG_NIMBLE_ROLE_OBSERVER (*CONFIG_BT_NIMBLE_ROLE_OBSERVER*)
- CONFIG_NIMBLE_ROLE_PERIPHERAL (*CONFIG_BT_NIMBLE_ROLE_PERIPHERAL*)
- CONFIG_NIMBLE_RPA_TIMEOUT (*CONFIG_BT_NIMBLE_RPA_TIMEOUT*)
- CONFIG_NIMBLE_SM_LEGACY (*CONFIG_BT_NIMBLE_SM_LEGACY*)
- CONFIG_NIMBLE_SM_SC (*CONFIG_BT_NIMBLE_SM_SC*)
- CONFIG_NIMBLE_SM_SC_DEBUG_KEYS (*CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS*)
- CONFIG_NIMBLE_SVC_GAP_APPEARANCE (*CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE*)
- CONFIG_NIMBLE_SVC_GAP_DEVICE_NAME (*CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME*)
- CONFIG_NIMBLE_TASK_STACK_SIZE (*CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE*)

- CONFIG_NO_BLOBS (*CONFIG_APP_NO_BLOBS*)
- **CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS (*CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES*)**
  - CONFIG_TWO_UNIVERSAL_MAC_ADDRESS
  - CONFIG_FOUR_UNIVERSAL_MAC_ADDRESS
- **CONFIG_OPTIMIZATION_ASSERTION_LEVEL (*CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*)**
  - CONFIG_OPTIMIZATION_ASSERTIONS_ENABLED
  - CONFIG_OPTIMIZATION_ASSERTIONS_SILENT
  - CONFIG_OPTIMIZATION_ASSERTIONS_DISABLED
- **CONFIG_OPTIMIZATION_COMPILER (*CONFIG_COMPILER_OPTIMIZATION*)**
  - CONFIG_OPTIMIZATION_LEVEL_DEBUG, CONFIG_COMPILER_OPTIMIZATION_LEVEL_DEBUG
  - CONFIG_OPTIMIZATION_LEVEL_RELEASE, CONFIG_COMPILER_OPTIMIZATION_LEVEL_RELEASE
- **CONFIG_OSI_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_OSI_TRACE_LEVEL*)**
  - CONFIG_OSI_TRACE_LEVEL_NONE
  - CONFIG_OSI_TRACE_LEVEL_ERROR
  - CONFIG_OSI_TRACE_LEVEL_WARNING
  - CONFIG_OSI_TRACE_LEVEL_API
  - CONFIG_OSI_TRACE_LEVEL_EVENT
  - CONFIG_OSI_TRACE_LEVEL_DEBUG
  - CONFIG_OSI_TRACE_LEVEL_VERBOSE
- CONFIG_OTA_ALLOW_HTTP (*CONFIG_ESP_HTTPS_OTA_ALLOW_HTTP*)
- **CONFIG_PAN_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_PAN_TRACE_LEVEL*)**
  - CONFIG_PAN_TRACE_LEVEL_NONE
  - CONFIG_PAN_TRACE_LEVEL_ERROR
  - CONFIG_PAN_TRACE_LEVEL_WARNING
  - CONFIG_PAN_TRACE_LEVEL_API
  - CONFIG_PAN_TRACE_LEVEL_EVENT
  - CONFIG_PAN_TRACE_LEVEL_DEBUG
  - CONFIG_PAN_TRACE_LEVEL_VERBOSE
- CONFIG_POST_EVENTS_FROM_IRAM_ISR (*CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR*)
- CONFIG_POST_EVENTS_FROM_ISR (*CONFIG_ESP_EVENT_POST_FROM_ISR*)
- CONFIG_PPP_CHAP_SUPPORT (*CONFIG_LWIP_PPP_CHAP_SUPPORT*)
- CONFIG_PPP_DEBUG_ON (*CONFIG_LWIP_PPP_DEBUG_ON*)
- CONFIG_PPP_MPPE_SUPPORT (*CONFIG_LWIP_PPP_MPPE_SUPPORT*)
- CONFIG_PPP_MSCHAP_SUPPORT (*CONFIG_LWIP_PPP_MSCHAP_SUPPORT*)
- CONFIG_PPP_NOTIFY_PHASE_SUPPORT (*CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT*)
- CONFIG_PPP_PAP_SUPPORT (*CONFIG_LWIP_PPP_PAP_SUPPORT*)
- CONFIG_PPP_SUPPORT (*CONFIG_LWIP_PPP_SUPPORT*)
- CONFIG_REDUCE_PHY_TX_POWER (*CONFIG_ESP_PHY_REDUCE_TX_POWER*)
- **CONFIG_RFCOMM_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL*)**
  - CONFIG_RFCOMM_TRACE_LEVEL_NONE
  - CONFIG_RFCOMM_TRACE_LEVEL_ERROR
  - CONFIG_RFCOMM_TRACE_LEVEL_WARNING
  - CONFIG_RFCOMM_TRACE_LEVEL_API
  - CONFIG_RFCOMM_TRACE_LEVEL_EVENT
  - CONFIG_RFCOMM_TRACE_LEVEL_DEBUG
  - CONFIG_RFCOMM_TRACE_LEVEL_VERBOSE
- **CONFIG_SCAN_DUPLICATE_TYPE (*CONFIG_BTDM_SCAN_DUPL_TYPE*)**
  - CONFIG_SCAN_DUPLICATE_BY_DEVICE_ADDR
  - CONFIG_SCAN_DUPLICATE_BY_ADV_DATA
  - CONFIG_SCAN_DUPLICATE_BY_ADV_DATA_AND_DEVICE_ADDR
- CONFIG_SEMIHOSTFS_MAX_MOUNT_POINTS (*CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS*)
- **CONFIG_SMP_INITIAL_TRACE_LEVEL (*CONFIG_BT_LOG_SMP_TRACE_LEVEL*)**
  - CONFIG_SMP_TRACE_LEVEL_NONE
  - CONFIG_SMP_TRACE_LEVEL_ERROR
  - CONFIG_SMP_TRACE_LEVEL_WARNING
  - CONFIG_SMP_TRACE_LEVEL_API

    - CONFIG_SMP_TRACE_LEVEL_EVENT
    - CONFIG_SMP_TRACE_LEVEL_DEBUG
    - CONFIG_SMP_TRACE_LEVEL_VERBOSE
- CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE (*CONFIG_BT_SMP_SLAVE_CON_PARAMS_UPD_ENABLE*)
- **CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS (*CONFIG_SPI_FLASH_DANGEROUS_WRITE*)**
    - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ABORTS
    - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_FAILS
    - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ALLOWED
- **CONFIG_STACK_CHECK_MODE (*CONFIG_COMPILER_STACK_CHECK_MODE*)**
    - CONFIG_STACK_CHECK_NONE
    - CONFIG_STACK_CHECK_NORM
    - CONFIG_STACK_CHECK_STRONG
    - CONFIG_STACK_CHECK_ALL
- CONFIG_SUPPORT_TERMIOS (*CONFIG_VFS_SUPPORT_TERMIOS*)
- CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT (*CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT*)
- CONFIG_SW_COEXIST_ENABLE (*CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE*)
- CONFIG_SYSTEM_EVENT_QUEUE_SIZE (*CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE*)
- CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE (*CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE*)
- CONFIG_SYSVIEW_BUF_WAIT_TMO (*CONFIG_APPTRACE_SV_BUF_WAIT_TMO*)
- CONFIG_SYSVIEW_ENABLE (*CONFIG_APPTRACE_SV_ENABLE*)
- CONFIG_SYSVIEW_EVT_IDLE_ENABLE (*CONFIG_APPTRACE_SV_EVT_IDLE_ENABLE*)
- CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE (*CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE*)
- CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE (*CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE*)
- CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE (*CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE*)
- CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE (*CONFIG_APPTRACE_SV_EVT_OVERFLOW_ENABLE*)
- CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE*)
- CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABL*
- CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_START_READY_EN*
- CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE*)
- CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_STOP_READY_ENAB*
- CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE (*CONFIG_APPTRACE_SV_EVT_TASK_TERMINATE_ENABLE*)
- CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE (*CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE*)
- CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE (*CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE*)
- CONFIG_SYSVIEW_MAX_TASKS (*CONFIG_APPTRACE_SV_MAX_TASKS*)
- **CONFIG_SYSVIEW_TS_SOURCE (*CONFIG_APPTRACE_SV_TS_SOURCE*)**
    - CONFIG_SYSVIEW_TS_SOURCE_CCOUNT
    - CONFIG_SYSVIEW_TS_SOURCE_ESP_TIMER
- CONFIG_TASK_WDT (*CONFIG_ESP_TASK_WDT_INIT*)
- CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0 (*CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0*)
- CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1 (*CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1*)
- CONFIG_TASK_WDT_PANIC (*CONFIG_ESP_TASK_WDT_PANIC*)
- CONFIG_TASK_WDT_TIMEOUT_S (*CONFIG_ESP_TASK_WDT_TIMEOUT_S*)
- CONFIG_TCPIP_RECVMBOX_SIZE (*CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*)
- **CONFIG_TCPIP_TASK_AFFINITY (*CONFIG_LWIP_TCPIP_TASK_AFFINITY*)**
    - CONFIG_TCPIP_TASK_AFFINITY_NO_AFFINITY
    - CONFIG_TCPIP_TASK_AFFINITY_CPU0
    - CONFIG_TCPIP_TASK_AFFINITY_CPU1
- CONFIG_TCPIP_TASK_STACK_SIZE (*CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*)
- CONFIG_TCP_MAXRTX (*CONFIG_LWIP_TCP_MAXRTX*)
- CONFIG_TCP_MSL (*CONFIG_LWIP_TCP_MSL*)
- CONFIG_TCP_MSS (*CONFIG_LWIP_TCP_MSS*)
- **CONFIG_TCP_OVERSIZE (*CONFIG_LWIP_TCP_OVERSIZE*)**
    - CONFIG_TCP_OVERSIZE_MSS
    - CONFIG_TCP_OVERSIZE_QUARTER_MSS
    - CONFIG_TCP_OVERSIZE_DISABLE
- CONFIG_TCP_QUEUE_OOSEQ (*CONFIG_LWIP_TCP_QUEUE_OOSEQ*)
- CONFIG_TCP_RECVMBOX_SIZE (*CONFIG_LWIP_TCP_RECVMBOX_SIZE*)

- CONFIG_TCP_SND_BUF_DEFAULT (*CONFIG_LWIP_TCP_SND_BUF_DEFAULT*)
- CONFIG_TCP_SYNMAXRTX (*CONFIG_LWIP_TCP_SYNMAXRTX*)
- CONFIG_TCP_WND_DEFAULT (*CONFIG_LWIP_TCP_WND_DEFAULT*)
- CONFIG_TIMER_QUEUE_LENGTH (*CONFIG_FREERTOS_TIMER_QUEUE_LENGTH*)
- CONFIG_TIMER_TASK_PRIORITY (*CONFIG_FREERTOS_TIMER_TASK_PRIORITY*)
- CONFIG_TIMER_TASK_STACK_DEPTH (*CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH*)
- CONFIG_TIMER_TASK_STACK_SIZE (*CONFIG_ESP_TIMER_TASK_STACK_SIZE*)
- CONFIG_UDP_RECVMBOX_SIZE (*CONFIG_LWIP_UDP_RECVMBOX_SIZE*)
- CONFIG_WARN_WRITE_STRINGS (*CONFIG_COMPILER_WARN_WRITE_STRINGS*)

2.8 Provisioning API

2.8.1 Protocol Communication

Overview

The Protocol Communication (protocomm) component manages secure sessions and provides the framework for multiple transports. The application can also use the protocomm layer directly to have application-specific extensions for the provisioning or non-provisioning use cases.

Following features are available for provisioning:

- Communication security at the application level
  - `protocomm_security0` (no security)
  - `protocomm_security1` (Curve25519 key exchange + AES-CTR encryption/decryption)
  - `protocomm_security2` (SRP6a-based key exchange + AES-GCM encryption/decryption)
- Proof-of-possession (support with protocomm_security1 only)
- Salt and Verifier (support with protocomm_security2 only)

Protocomm internally uses protobuf (protocol buffers) for secure session establishment. Users can choose to implement their own security (even without using protobuf). Protocomm can also be used without any security layer.

Protocomm provides the framework for various transports:

- Bluetooth LE
- Wi-Fi (SoftAP + HTTPD)
- Console, in which case the handler invocation is automatically taken care of on the device side. See Transport Examples below for code snippets.

Note that for protocomm_security1 and protocomm_security2, the client still needs to establish sessions by performing the two-way handshake. See *Unified Provisioning* for more details about the secure handshake logic.

Enabling Protocomm Security Version

The protocomm component provides a project configuration menu to enable/disable support of respective security versions. The respective configuration options are as follows:

- Support `protocomm_security0`, with no security: *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0*, this option is enabled by default.
- Support `protocomm_security1` with Curve25519 key exchange + AES-CTR encryption/decryption: *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1*, this option is enabled by default.

- Support `protocomm_security2` with SRP6a-based key exchange + AES-GCM encryption/decryption: *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2*.

Note: Enabling multiple security versions at once offers the ability to control them dynamically but also increases the firmware size.

SoftAP + HTTP Transport Example with Security 2

For sample usage, see wifi_provisioning/src/scheme_softap.c.

```
/* The endpoint handler to be registered with protocomm. This simply echoes back␣
↪the received data. */
esp_err_t echo_req_handler (uint32_t session_id,
                            const uint8_t *inbuf, ssize_t inlen,
                            uint8_t **outbuf, ssize_t *outlen,
                            void *priv_data)
{
    /* Session ID may be used for persistence. */
    printf("Session ID : %d", session_id);

    /* Echo back the received data. */
    *outlen = inlen;            /* Output the data length updated. */
    *outbuf = malloc(inlen);    /* This is to be deallocated outside. */
    memcpy(*outbuf, inbuf, inlen);

    /* Private data that was passed at the time of endpoint creation. */
    uint32_t *priv = (uint32_t *) priv_data;
    if (priv) {
        printf("Private data : %d", *priv);
    }

    return ESP_OK;
}

static const char sec2_salt[] = {0xf7, 0x5f, 0xe2, 0xbe, 0xba, 0x7c, 0x81, 0xcd};
static const char sec2_verifier[] = {0xbf, 0x86, 0xce, 0x63, 0x8a, 0xbb, 0x7e,␣
↪0x2f, 0x38, 0xa8, 0x19, 0x1b, 0x35,
    0xc9, 0xe3, 0xbe, 0xc3, 0x2b, 0x45, 0xee, 0x10, 0x74, 0x22, 0x1a, 0x95, 0xbe,␣
↪0x62, 0xf7, 0x0c, 0x65, 0x83, 0x50,
    0x08, 0xef, 0xaf, 0xa5, 0x94, 0x4b, 0xcb, 0xe1, 0xce, 0x59, 0x2a, 0xe8, 0x7b,␣
↪0x27, 0xc8, 0x72, 0x26, 0x71, 0xde,
    0xb2, 0xf2, 0x80, 0x02, 0xdd, 0x11, 0xf0, 0x38, 0x0e, 0x95, 0x25, 0x00, 0xcf,␣
↪0xb3, 0x3f, 0xf0, 0x73, 0x2a, 0x25,
    0x03, 0xe8, 0x51, 0x72, 0xef, 0x6d, 0x3e, 0x14, 0xb9, 0x2e, 0x9f, 0x2a, 0x90,␣
↪0x9e, 0x26, 0xb6, 0x3e, 0xc7, 0xe4,
    0x9f, 0xe3, 0x20, 0xce, 0x28, 0x7c, 0xbf, 0x89, 0x50, 0xc9, 0xb6, 0xec, 0xdd,␣
↪0x81, 0x18, 0xf1, 0x1a, 0xd9, 0x7a,
    0x21, 0x99, 0xf1, 0xee, 0x71, 0x2f, 0xcc, 0x93, 0x16, 0x34, 0x0c, 0x79, 0x46,␣
↪0x23, 0xe4, 0x32, 0xec, 0x2d, 0x9e,
    0x18, 0xa6, 0xb9, 0xbb, 0x0a, 0xcf, 0xc4, 0xa8, 0x32, 0xc0, 0x1c, 0x32, 0xa3,␣
↪0x97, 0x66, 0xf8, 0x30, 0xb2, 0xda,
    0xf9, 0x8d, 0xc3, 0x72, 0x72, 0x5f, 0xe5, 0xee, 0xc3, 0x5c, 0x24, 0xc8, 0xdd,␣
↪0x54, 0x49, 0xfc, 0x12, 0x91, 0x81,
    0x9c, 0xc3, 0xac, 0x64, 0x5e, 0xd6, 0x41, 0x88, 0x2f, 0x23, 0x66, 0xc8, 0xac,␣
↪0xb0, 0x35, 0x0b, 0xf6, 0x9c, 0x88,
    0x6f, 0xac, 0xe1, 0xf4, 0xca, 0xc9, 0x07, 0x04, 0x11, 0xda, 0x90, 0x42, 0xa9,␣
↪0xf1, 0x97, 0x3d, 0x94, 0x65, 0xe4,
    0xfb, 0x52, 0x22, 0x3b, 0x7a, 0x7b, 0x9e, 0xe9, 0xee, 0x1c, 0x44, 0xd0, 0x73,␣
↪0x72, 0x2a, 0xca, 0x85, 0x19, 0x4a,
```

(continues on next page)

(continued from previous page)

```
    0x60, 0xce, 0x0a, 0xc8, 0x7d, 0x57, 0xa4, 0xf8, 0x77, 0x22, 0xc1, 0xa5, 0xfa,␣
↪0xfb, 0x7b, 0x91, 0x3b, 0xfe, 0x87,
    0x5f, 0xfe, 0x05, 0xd2, 0xd6, 0xd3, 0x74, 0xe5, 0x2e, 0x68, 0x79, 0x34, 0x70,␣
↪0x40, 0x12, 0xa8, 0xe1, 0xb4, 0x6c,
    0xaa, 0x46, 0x73, 0xcd, 0x8d, 0x17, 0x72, 0x67, 0x32, 0x42, 0xdc, 0x10, 0xd3,␣
↪0x71, 0x7e, 0x8b, 0x00, 0x46, 0x9b,
    0x0a, 0xe9, 0xb4, 0x0f, 0xeb, 0x70, 0x52, 0xdd, 0x0a, 0x1c, 0x7e, 0x2e, 0xb0,␣
↪0x61, 0xa6, 0xe1, 0xa3, 0x34, 0x4b,
    0x2a, 0x3c, 0xc4, 0x5d, 0x42, 0x05, 0x58, 0x25, 0xd3, 0xca, 0x96, 0x5c, 0xb9,␣
↪0x52, 0xf9, 0xe9, 0x80, 0x75, 0x3d,
    0xc8, 0x9f, 0xc7, 0xb2, 0xaa, 0x95, 0x2e, 0x76, 0xb3, 0xe1, 0x48, 0xc1, 0x0a,␣
↪0xa1, 0x0a, 0xe8, 0xaf, 0x41, 0x28,
    0xd2, 0x16, 0xe1, 0xa6, 0xd0, 0x73, 0x51, 0x73, 0x79, 0x98, 0xd9, 0xb9, 0x00,␣
↪0x50, 0xa2, 0x4d, 0x99, 0x18, 0x90,
    0x70, 0x27, 0xe7, 0x8d, 0x56, 0x45, 0x34, 0x1f, 0xb9, 0x30, 0xda, 0xec, 0x4a,␣
↪0x08, 0x27, 0x9f, 0xfa, 0x59, 0x2e,
    0x36, 0x77, 0x00, 0xe2, 0xb6, 0xeb, 0xd1, 0x56, 0x50, 0x8e};

/* The example function for launching a protocomm instance over HTTP. */
protocomm_t *start_pc()
{
    protocomm_t *pc = protocomm_new();


    /* Config for protocomm_httpd_start(). */
    protocomm_httpd_config_t pc_config = {
        .data = {
        .config = PROTOCOMM_HTTPD_DEFAULT_CONFIG()
        }
    };

    /* Start the protocomm server on top of HTTP. */
    protocomm_httpd_start(pc, &pc_config);

    /* Create Security2 params object from salt and verifier. It must be valid␣
↪throughout the scope of protocomm endpoint. This does not need to be static, i.e.
↪, could be dynamically allocated and freed at the time of endpoint removal. */
    const static protocomm_security2_params_t sec2_params = {
        .salt = (const uint8_t *) salt,
        .salt_len = sizeof(salt),
        .verifier = (const uint8_t *) verifier,
        .verifier_len = sizeof(verifier),
    };

    /* Set security for communication at the application level. Just like for␣
↪request handlers, setting security creates an endpoint and registers the handler␣
↪provided by protocomm_security1. One can similarly use protocomm_security0. Only␣
↪one type of security can be set for a protocomm instance at a time. */
    protocomm_set_security(pc, "security_endpoint", &protocomm_security2, &sec2_
↪params);

    /* Private data passed to the endpoint must be valid throughout the scope of␣
↪protocomm endpoint. This need not be static, i.e., could be dynamically␣
↪allocated and freed at the time of endpoint removal. */
    static uint32_t priv_data = 1234;

    /* Add a new endpoint for the protocomm instance, identified by a unique name,␣
↪and register a handler function along with the private data to be passed at the␣
↪time of handler execution. Multiple endpoints can be added as long as they are␣
↪identified by unique names. */
    protocomm_add_endpoint(pc, "echo_req_endpoint",
```

(continues on next page)

(continued from previous page)

```
                            echo_req_handler, (void *) &priv_data);
    return pc;
}

/* The example function for stopping a protocomm instance. */
void stop_pc(protocomm_t *pc)
{
    /* Remove the endpoint identified by its unique name. */
    protocomm_remove_endpoint(pc, "echo_req_endpoint");

    /* Remove the security endpoint identified by its name. */
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop the HTTP server. */
    protocomm_httpd_stop(pc);

    /* Delete, namely deallocate the protocomm instance. */
    protocomm_delete(pc);
}
```

SoftAP + HTTP Transport Example with Security 1

For sample usage, see wifi_provisioning/src/scheme_softap.c.

```
/* The endpoint handler to be registered with protocomm. This simply echoes back␣
↪the received data. */
esp_err_t echo_req_handler (uint32_t session_id,
                            const uint8_t *inbuf, ssize_t inlen,
                            uint8_t **outbuf, ssize_t *outlen,
                            void *priv_data)
{
    /* Session ID may be used for persistence. */
    printf("Session ID : %d", session_id);

    /* Echo back the received data. */
    *outlen = inlen;            /* Output the data length updated. */
    *outbuf = malloc(inlen);    /* This is to be deallocated outside. */
    memcpy(*outbuf, inbuf, inlen);

    /* Private data that was passed at the time of endpoint creation. */
    uint32_t *priv = (uint32_t *) priv_data;
    if (priv) {
        printf("Private data : %d", *priv);
    }

    return ESP_OK;
}

/* The example function for launching a protocomm instance over HTTP. */
protocomm_t *start_pc(const char *pop_string)
{
    protocomm_t *pc = protocomm_new();


    /* Config for protocomm_httpd_start(). */
    protocomm_httpd_config_t pc_config = {
        .data = {
        .config = PROTOCOMM_HTTPD_DEFAULT_CONFIG()
        }
    };
```

(continues on next page)

(continued from previous page)

```
    /* Start the protocomm server on top of HTTP. */
    protocomm_httpd_start(pc, &pc_config);

    /* Create security1 params object from pop_string. It must be valid throughout␣
↪the scope of protocomm endpoint. This need not be static, i.e., could be␣
↪dynamically allocated and freed at the time of endpoint removal. */
    const static protocomm_security1_params_t sec1_params = {
        .data = (const uint8_t *) strdup(pop_string),
        .len = strlen(pop_string)
    };

    /* Set security for communication at the application level. Just like for␣
↪request handlers, setting security creates an endpoint and registers the handler␣
↪provided by protocomm_security1. One can similarly use protocomm_security0. Only␣
↪one type of security can be set for a protocomm instance at a time. */
    protocomm_set_security(pc, "security_endpoint", &protocomm_security1, &sec1_
↪params);

    /* Private data passed to the endpoint must be valid throughout the scope of␣
↪protocomm endpoint. This need not be static, i.e., could be dynamically␣
↪allocated and freed at the time of endpoint removal. */
    static uint32_t priv_data = 1234;

    /* Add a new endpoint for the protocomm instance identified by a unique name,␣
↪and register a handler function along with the private data to be passed at the␣
↪time of handler execution. Multiple endpoints can be added as long as they are␣
↪identified by unique names. */
    protocomm_add_endpoint(pc, "echo_req_endpoint",
                           echo_req_handler, (void *) &priv_data);
    return pc;
}

/* The example function for stopping a protocomm instance. */
void stop_pc(protocomm_t *pc)
{
    /* Remove the endpoint identified by its unique name. */
    protocomm_remove_endpoint(pc, "echo_req_endpoint");

    /* Remove the security endpoint identified by its name. */
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop the HTTP server. */
    protocomm_httpd_stop(pc);

    /* Delete, namely deallocate the protocomm instance. */
    protocomm_delete(pc);
}
```

Bluetooth LE Transport Example with Security 0

For sample usage, see wifi_provisioning/src/scheme_ble.c.

```
/* The example function for launching a secure protocomm instance over Bluetooth␣
↪LE. */
protocomm_t *start_pc()
{
    protocomm_t *pc = protocomm_new();

    /* Endpoint UUIDs */
```

(continues on next page)

(continued from previous page)

```
    protocomm_ble_name_uuid_t nu_lookup_table[] = {
        {"security_endpoint", 0xFF51},
        {"echo_req_endpoint", 0xFF52}
    };

    /* Config for protocomm_ble_start(). */
    protocomm_ble_config_t config = {
        .service_uuid = {
            /* LSB <---------------------------------------
            * ---------------------------------------> MSB */
            0xfb, 0x34, 0x9b, 0x5f, 0x80, 0x00, 0x00, 0x80,
            0x00, 0x10, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00,
        },
        .nu_lookup_count = sizeof(nu_lookup_table)/sizeof(nu_lookup_table[0]),
        .nu_lookup = nu_lookup_table
    };

    /* Start protocomm layer on top of Bluetooth LE. */
    protocomm_ble_start(pc, &config);

    /* For protocomm_security0, Proof of Possession is not used, and can be kept␣
↪NULL. */
    protocomm_set_security(pc, "security_endpoint", &protocomm_security0, NULL);
    protocomm_add_endpoint(pc, "echo_req_endpoint", echo_req_handler, NULL);
    return pc;
}

/* The example function for stopping a protocomm instance. */
void stop_pc(protocomm_t *pc)
{
    protocomm_remove_endpoint(pc, "echo_req_endpoint");
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop the Bluetooth LE protocomm service. */
    protocomm_ble_stop(pc);

    protocomm_delete(pc);
}
```

API Reference

Header File

- components/protocomm/include/common/protocomm.h

Functions

protocomm_t ***protocomm_new**(void)

Create a new protocomm instance.

This API will return a new dynamically allocated protocomm instance with all elements of the protocomm_t structure initialized to NULL.

Returns

- protocomm_t* : On success
- NULL : No memory for allocating new instance

void **protocomm_delete**(*protocomm_t* *pc)

Delete a protocomm instance.

This API will deallocate a protocomm instance that was created using `protocomm_new()`.

Parameters **pc** –**[in]** Pointer to the protocomm instance to be deleted

esp_err_t **protocomm_add_endpoint** (*protocomm_t* *pc, const char *ep_name, *protocomm_req_handler_t* h, void *priv_data)

Add endpoint request handler for a protocomm instance.

This API will bind an endpoint handler function to the specified endpoint name, along with any private data that needs to be pass to the handler at the time of call.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.
- This function internally calls the registered `add_endpoint()` function of the selected transport which is a member of the protocomm_t instance structure.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **ep_name** –**[in]** Endpoint identifier(name) string
- **h** –**[in]** Endpoint handler function
- **priv_data** –**[in]** Pointer to private data to be passed as a parameter to the handler function on call. Pass NULL if not needed.

Returns

- ESP_OK : Success
- ESP_FAIL : Error adding endpoint / Endpoint with this name already exists
- ESP_ERR_NO_MEM : Error allocating endpoint resource
- ESP_ERR_INVALID_ARG : Null instance/name/handler arguments

esp_err_t **protocomm_remove_endpoint** (*protocomm_t* *pc, const char *ep_name)

Remove endpoint request handler for a protocomm instance.

This API will remove a registered endpoint handler identified by an endpoint name.

Note:

- This function internally calls the registered `remove_endpoint()` function which is a member of the protocomm_t instance structure.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **ep_name** –**[in]** Endpoint identifier(name) string

Returns

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn’ t exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

esp_err_t **protocomm_open_session** (*protocomm_t* *pc, uint32_t session_id)

Allocates internal resources for new transport session.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **session_id** –**[in]** Unique ID for a communication session

Returns

- ESP_OK : Request handled successfully
- ESP_ERR_NO_MEM : Error allocating internal resource
- ESP_ERR_INVALID_ARG : Null instance/name arguments

esp_err_t **protocomm_close_session** (*protocomm_t* *pc, uint32_t session_id)

Frees internal resources used by a transport session.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **session_id** –**[in]** Unique ID for a communication session

Returns

- ESP_OK : Request handled successfully
- ESP_ERR_INVALID_ARG : Null instance/name arguments

esp_err_t **protocomm_req_handle** (*protocomm_t* *pc, const char *ep_name, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen)

Calls the registered handler of an endpoint session for processing incoming data and generating the response.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.
- Resulting output buffer must be deallocated by the caller.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **ep_name** –**[in]** Endpoint identifier(name) string
- **session_id** –**[in]** Unique ID for a communication session
- **inbuf** –**[in]** Input buffer contains input request data which is to be processed by the registered handler
- **inlen** –**[in]** Length of the input buffer
- **outbuf** –**[out]** Pointer to internally allocated output buffer, where the resulting response data output from the registered handler is to be stored
- **outlen** –**[out]** Buffer length of the allocated output buffer

Returns

- ESP_OK : Request handled successfully
- ESP_FAIL : Internal error in execution of registered handler
- ESP_ERR_NO_MEM : Error allocating internal resource
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn’ t exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

esp_err_t **protocomm_set_security** (*protocomm_t* *pc, const char *ep_name, const *protocomm_security_t* *sec, const void *sec_params)

Add endpoint security for a protocomm instance.

This API will bind a security session establisher to the specified endpoint name, along with any proof of possession that may be required for authenticating a session client.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

- The choice of security can be any `protocomm_security_t` instance. Choices `protocomm_security0` and `protocomm_security1` and `protocomm_security2` are readily available.

Parameters

- **`pc`** –**[in]** Pointer to the protocomm instance
- **`ep_name`** –**[in]** Endpoint identifier(name) string
- **`sec`** –**[in]** Pointer to endpoint security instance
- **`sec_params`** –**[in]** Pointer to security params (NULL if not needed) The pointer should contain the security params struct of appropriate security version. For protocomm security version 1 and 2 sec_params should contain pointer to struct of type protocomm_security1_params_t and protocmm_security2_params_t respectively. The contents of this pointer must be valid till the security session has been running and is not closed.

Returns

- ESP_OK : Success
- ESP_FAIL : Error adding endpoint / Endpoint with this name already exists
- ESP_ERR_INVALID_STATE : Security endpoint already set
- ESP_ERR_NO_MEM : Error allocating endpoint resource
- ESP_ERR_INVALID_ARG : Null instance/name/handler arguments

esp_err_t **`protocomm_unset_security`**(*protocomm_t* *pc, const char *ep_name)

Remove endpoint security for a protocomm instance.

This API will remove a registered security endpoint identified by an endpoint name.

Parameters

- **`pc`** –**[in]** Pointer to the protocomm instance
- **`ep_name`** –**[in]** Endpoint identifier(name) string

Returns

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn’ t exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

esp_err_t **`protocomm_set_version`**(*protocomm_t* *pc, const char *ep_name, const char *version)

Set endpoint for version verification.

This API can be used for setting an application specific protocol version which can be verified by clients through the endpoint.

Note:

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

Parameters

- **`pc`** –**[in]** Pointer to the protocomm instance
- **`ep_name`** –**[in]** Endpoint identifier(name) string
- **`version`** –**[in]** Version identifier(name) string

Returns

- ESP_OK : Success
- ESP_FAIL : Error adding endpoint / Endpoint with this name already exists
- ESP_ERR_INVALID_STATE : Version endpoint already set
- ESP_ERR_NO_MEM : Error allocating endpoint resource
- ESP_ERR_INVALID_ARG : Null instance/name/handler arguments

esp_err_t **`protocomm_unset_version`**(*protocomm_t* *pc, const char *ep_name)

Remove version verification endpoint from a protocomm instance.

This API will remove a registered version endpoint identified by an endpoint name.

Parameters

- **pc** –**[in]** Pointer to the protocomm instance
- **ep_name** –**[in]** Endpoint identifier(name) string

Returns

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn't exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

Type Definitions

typedef *esp_err_t* (***protocomm_req_handler_t**)(uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen, void *priv_data)

Function prototype for protocomm endpoint handler.

typedef struct protocomm **protocomm_t**

This structure corresponds to a unique instance of protocomm returned when the API `protocomm_new()` is called. The remaining Protocomm APIs require this object as the first parameter.

Note: Structure of the protocomm object is kept private

Header File

- components/protocomm/include/security/protocomm_security.h

Structures

struct **protocomm_security1_params**

Protocomm Security 1 parameters: Proof Of Possession.

Public Members

const uint8_t ***data**

Pointer to buffer containing the proof of possession data

uint16_t **len**

Length (in bytes) of the proof of possession data

struct **protocomm_security2_params**

Protocomm Security 2 parameters: Salt and Verifier.

Public Members

const char ***salt**

Pointer to the buffer containing the salt

uint16_t **salt_len**

Length (in bytes) of the salt

const char ***verifier**

Pointer to the buffer containing the verifier

uint16_t **verifier_len**

Length (in bytes) of the verifier

struct **protocomm_security**

Protocomm security object structure.

The member functions are used for implementing secure protocomm sessions.

Note: This structure should not have any dynamic members to allow re-entrancy

Public Members

int **ver**

Unique version number of security implementation

esp_err_t (***init**)(*protocomm_security_handle_t* *handle)

Function for initializing/allocating security infrastructure

esp_err_t (***cleanup**)(*protocomm_security_handle_t* handle)

Function for deallocating security infrastructure

esp_err_t (***new_transport_session**)(*protocomm_security_handle_t* handle, uint32_t session_id)

Starts new secure transport session with specified ID

esp_err_t (***close_transport_session**)(*protocomm_security_handle_t* handle, uint32_t session_id)

Closes a secure transport session with specified ID

esp_err_t (***security_req_handler**)(*protocomm_security_handle_t* handle, const void *sec_params, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen, void *priv_data)

Handler function for authenticating connection request and establishing secure session

esp_err_t (***encrypt**)(*protocomm_security_handle_t* handle, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen)

Function which implements the encryption algorithm

esp_err_t (***decrypt**)(*protocomm_security_handle_t* handle, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen)

Function which implements the decryption algorithm

Type Definitions

typedef struct *protocomm_security1_params* **protocomm_security1_params_t**

Protocomm Security 1 parameters: Proof Of Possession.

typedef *protocomm_security1_params_t* **protocomm_security_pop_t**

typedef struct *protocomm_security2_params* **protocomm_security2_params_t**

Protocomm Security 2 parameters: Salt and Verifier.

typedef void ***protocomm_security_handle_t**

typedef struct *protocomm_security* **protocomm_security_t**

Protocomm security object structure.

The member functions are used for implementing secure protocomm sessions.

Note: This structure should not have any dynamic members to allow re-entrancy

---

Header File

- components/protocomm/include/security/protocomm_security0.h

Header File

- components/protocomm/include/security/protocomm_security1.h

Header File

- components/protocomm/include/transports/protocomm_httpd.h

Functions

esp_err_t **protocomm_httpd_start** (*protocomm_t* *pc, const *protocomm_httpd_config_t* *config)

Start HTTPD protocomm transport.

This API internally creates a framework to allow endpoint registration and security configuration for the protocomm.

Note: This is a singleton. ie. Protocomm can have multiple instances, but only one instance can be bound to an HTTP transport layer.

Parameters

- **pc** –**[in]** Protocomm instance pointer obtained from protocomm_new()
- **config** –**[in]** Pointer to config structure for initializing HTTP server

Returns

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_SUPPORTED : Transport layer bound to another protocomm instance
- ESP_ERR_INVALID_STATE : Transport layer already bound to this protocomm instance
- ESP_ERR_NO_MEM : Memory allocation for server resource failed
- ESP_ERR_HTTPD_* : HTTP server error on start

esp_err_t **protocomm_httpd_stop** (*protocomm_t* *pc)

Stop HTTPD protocomm transport.

This API cleans up the HTTPD transport protocomm and frees all the handlers registered with the protocomm.

Parameters **pc** –**[in]** Same protocomm instance that was passed to protocomm_httpd_start()

Returns

- ESP_OK : Success

- ESP_ERR_INVALID_ARG : Null / incorrect protocomm instance pointer

Unions

union **protocomm_httpd_config_data_t**

#include <protocomm_httpd.h> Protocomm HTTPD Configuration Data

Public Members

void ***handle**

HTTP Server Handle, if ext_handle_provided is set to true

protocomm_http_server_config_t **config**

HTTP Server Configuration, if a server is not already active

Structures

struct **protocomm_http_server_config_t**

Config parameters for protocomm HTTP server.

Public Members

uint16_t **port**

Port on which the HTTP server will listen

size_t **stack_size**

Stack size of server task, adjusted depending upon stack usage of endpoint handler

unsigned **task_priority**

Priority of server task

struct **protocomm_httpd_config_t**

Config parameters for protocomm HTTP server.

Public Members

bool **ext_handle_provided**

Flag to indicate of an external HTTP Server Handle has been provided. In such as case, protocomm will use the same HTTP Server and not start a new one internally.

protocomm_httpd_config_data_t **data**

Protocomm HTTPD Configuration Data

Macros

PROTOCOMM_HTTPD_DEFAULT_CONFIG()

Header File

- components/protocomm/include/transports/protocomm_ble.h

Functions

esp_err_t **protocomm_ble_start** (*protocomm_t* *pc, const *protocomm_ble_config_t* *config)

Start Bluetooth Low Energy based transport layer for provisioning.

Initialize and start required BLE service for provisioning. This includes the initialization for characteristics/service for BLE.

Parameters
- **pc** –**[in]** Protocomm instance pointer obtained from protocomm_new()
- **config** –**[in]** Pointer to config structure for initializing BLE

Returns
- ESP_OK : Success
- ESP_FAIL : Simple BLE start error
- ESP_ERR_NO_MEM : Error allocating memory for internal resources
- ESP_ERR_INVALID_STATE : Error in ble config
- ESP_ERR_INVALID_ARG : Null arguments

esp_err_t **protocomm_ble_stop** (*protocomm_t* *pc)

Stop Bluetooth Low Energy based transport layer for provisioning.

Stops service/task responsible for BLE based interactions for provisioning

---

Note: You might want to optionally reclaim memory from Bluetooth. Refer to the documentation of `esp_bt_mem_release` in that case.

Parameters **pc** –**[in]** Same protocomm instance that was passed to protocomm_ble_start()

Returns
- ESP_OK : Success
- ESP_FAIL : Simple BLE stop error
- ESP_ERR_INVALID_ARG : Null / incorrect protocomm instance

Structures

struct **name_uuid**

This structure maps handler required by protocomm layer to UUIDs which are used to uniquely identify BLE characteristics from a smartphone or a similar client device.

Public Members

const char ***name**

Name of the handler, which is passed to protocomm layer

uint16_t **uuid**

UUID to be assigned to the BLE characteristic which is mapped to the handler

struct **protocomm_ble_config**

Config parameters for protocomm BLE service.

Public Members

char **device_name**[MAX_BLE_DEVNAME_LEN + 1]

BLE device name being broadcast at the time of provisioning

uint8_t **service_uuid**[BLE_UUID128_VAL_LENGTH]

128 bit UUID of the provisioning service

uint8_t ***manufacturer_data**

BLE device manufacturer data pointer in advertisement

ssize_t **manufacturer_data_len**

BLE device manufacturer data length in advertisement

ssize_t **nu_lookup_count**

Number of entries in the Name-UUID lookup table

protocomm_ble_name_uuid_t ***nu_lookup**

Pointer to the Name-UUID lookup table

unsigned **ble_bonding**

BLE bonding

unsigned **ble_sm_sc**

BLE security flag

unsigned **ble_link_encryption**

BLE security flag

Macros

MAX_BLE_DEVNAME_LEN

BLE device name cannot be larger than this value 31 bytes (max scan response size) - 1 byte (length) - 1 byte (type) = 29 bytes

BLE_UUID128_VAL_LENGTH

MAX_BLE_MANUFACTURER_DATA_LEN

Theoretically, the limit for max manufacturer length remains same as BLE device name i.e. 31 bytes (max scan response size) - 1 byte (length) - 1 byte (type) = 29 bytes However, manufacturer data goes along with BLE device name in scan response. So, it is important to understand the actual length should be smaller than (29 - (BLE device name length) - 2).

Type Definitions

typedef struct *name_uuid* **protocomm_ble_name_uuid_t**

This structure maps handler required by protocomm layer to UUIDs which are used to uniquely identify BLE characteristics from a smartphone or a similar client device.

typedef struct *protocomm_ble_config* **protocomm_ble_config_t**

Config parameters for protocomm BLE service.

2.8.2 Unified Provisioning

Overview

The unified provisioning support in the ESP-IDF provides an extensible mechanism to the developers to configure the device with the Wi-Fi credentials and/or other custom configuration using various transports and different security schemes. Depending on the use case, it provides a complete and ready solution for Wi-Fi network provisioning along with example iOS and Android applications. The developers can choose to extend the device-side and phone-app side implementations to accommodate their requirements for sending additional configuration data. The followings are the important features of this implementation:

1. **Extensible Protocol**

The protocol is completely flexible and it offers the ability for the developers to send custom configuration in the provisioning process. The data representation is also left to the application to decide.

2. **Transport Flexibility**

The protocol can work on Wi-Fi (SoftAP + HTTP server) or on Bluetooth LE as a transport protocol. The framework provides an ability to add support for any other transport easily as long as command-response behavior can be supported on the transport.

3. **Security Scheme Flexibility**

It is understood that each use case may require different security scheme to secure the data that is exchanged in the provisioning process. Some applications may work with SoftAP that is WPA2 protected or Bluetooth LE with the "just-works" security. Or the applications may consider the transport to be insecure and may want application-level security. The unified provisioning framework allows the application to choose the security as deemed suitable.

4. **Compact Data Representation**

The protocol uses Google Protobufs as a data representation for session setup and Wi-Fi provisioning. They provide a compact data representation and ability to parse the data in multiple programming languages in native format. Please note that this data representation is not forced on application-specific data and the developers may choose the representation of their choice.

Typical Provisioning Process

Deciding on Transport

The unified provisioning subsystem supports Wi-Fi (SoftAP+HTTP server) and Bluetooth LE (GATT based) transport schemes. The following points need to be considered while selecting the best possible transport for provisioning:

1. The Bluetooth LE-based transport has the advantage of maintaining an intact communication channel between the device and the client during the provisioning, which ensures reliable provisioning feedback.
2. The Bluetooth LE-based provisioning implementation makes the user experience better from the phone apps as on Android and iOS both, the phone app can discover and connect to the device without requiring the user to go out of the phone app.
3. However, the Bluetooth LE transport consumes about 110 KB memory at runtime. If the product does not use the Bluetooth LE or Bluetooth functionality after provisioning is done, almost all the memory can be reclaimed and added into the heap.
4. The SoftAP-based transport is highly interoperable. However, there are a few considerations:
    - The device uses the same radio to host the SoftAP and also to connect to the configured AP. Since these could potentially be on different channels, it may cause connection status updates not to be reliably received by the phone
    - The phone (client) has to disconnect from its current AP in order to connect to the SoftAP. The original network will get restored only when the provisioning process is complete, and the softAP is taken down.
5. The SoftAP transport does not require much additional memory for the Wi-Fi use cases.

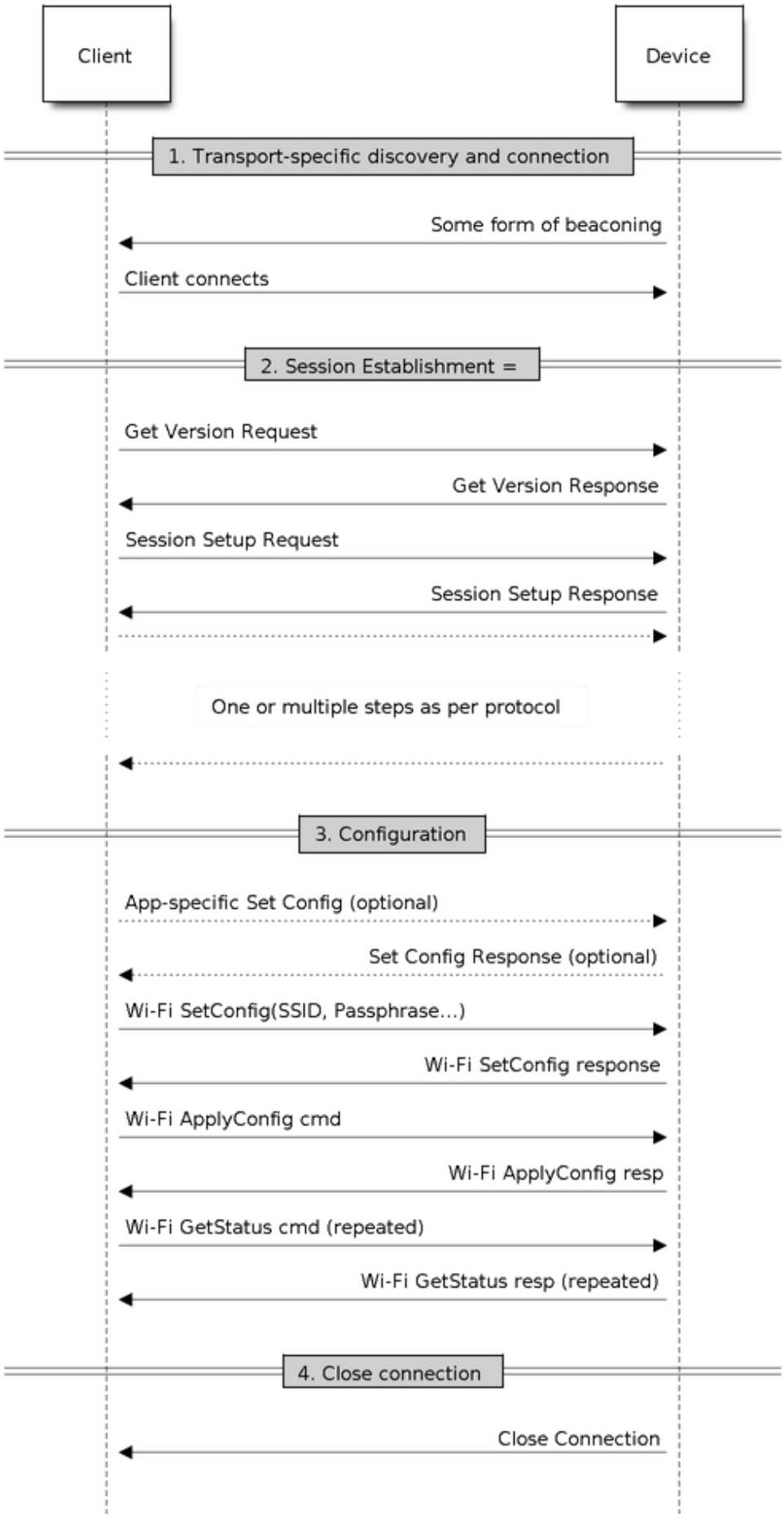


Fig. 28: Typical Provisioning Process

6. The SoftAP-based provisioning requires the phone-app user to go to `System Settings` to connect to the Wi-Fi network hosted by the device in the iOS system. The discovery (scanning) as well as connection APIs are not available for the iOS applications.

Deciding on Security

Depending on the transport and other constraints, the security scheme needs to be selected by the application developers. The following considerations need to be given from the provisioning-security perspective:

1. The configuration data sent from the client to the device and the response have to be secured.
2. The client should authenticate the device that it is connected to.
3. The device manufacturer may choose proof-of-possession (PoP), a unique per-device secret to be entered on the provisioning client as a security measure to make sure that only the user can provision the device in their possession.

There are two levels of security schemes, of which the developer may select one or a combination, depending on requirements.

1. **Transport Security**

For SoftAP provisioning, developers may choose WPA2-protected security with unique per-device passphrase. Unique per-device passphrase can also act as a proof-of-possession. For Bluetooth LE, the "just-works" security can be used as a transport-level security after assessing its provided level of security.

2. **Application Security**

The unified provisioning subsystem provides the application-level security (*Security 1 Scheme*) that provides data protection and authentication through PoP, if the application does not use the transport-level security, or if the transport-level security is not sufficient for the use case.

Device Discovery

The advertisement and device discovery is left to the application and depending on the protocol chosen, the phone apps and device-firmware application can choose appropriate method for advertisement and discovery.

For the SoftAP+HTTP transport, typically the SSID (network name) of the AP hosted by the device can be used for discovery.

For the Bluetooth LE transport, device name or primary service included in the advertisement or a combination of both can be used for discovery.

Architecture

The below diagram shows the architecture of unified provisioning:

It relies on the base layer called *Protocol Communication* (protocomm) which provides a framework for security schemes and transport mechanisms. The Wi-Fi Provisioning layer uses protocomm to provide simple callbacks to the application for setting the configuration and getting the Wi-Fi status. The application has control over implementation of these callbacks. In addition, the application can directly use protocomm to register custom handlers.

The application creates a protocomm instance which is mapped to a specific transport and specific security scheme. Each transport in the protocomm has a concept of an "end-point" which corresponds to the logical channel for communication for specific type of information. For example, security handshake happens on a different endpoint from the Wi-Fi configuration endpoint. Each end-point is identified using a string and depending on the transport internal representation of the end-point changes. In case of the SoftAP+HTTP transport, the end-point corresponds to URI, whereas in case of Bluetooth LE, the end-point corresponds to the GATT characteristic with specific UUID. Developers can create custom end-points and implement handler for the data that is received or sent over the same end-point.

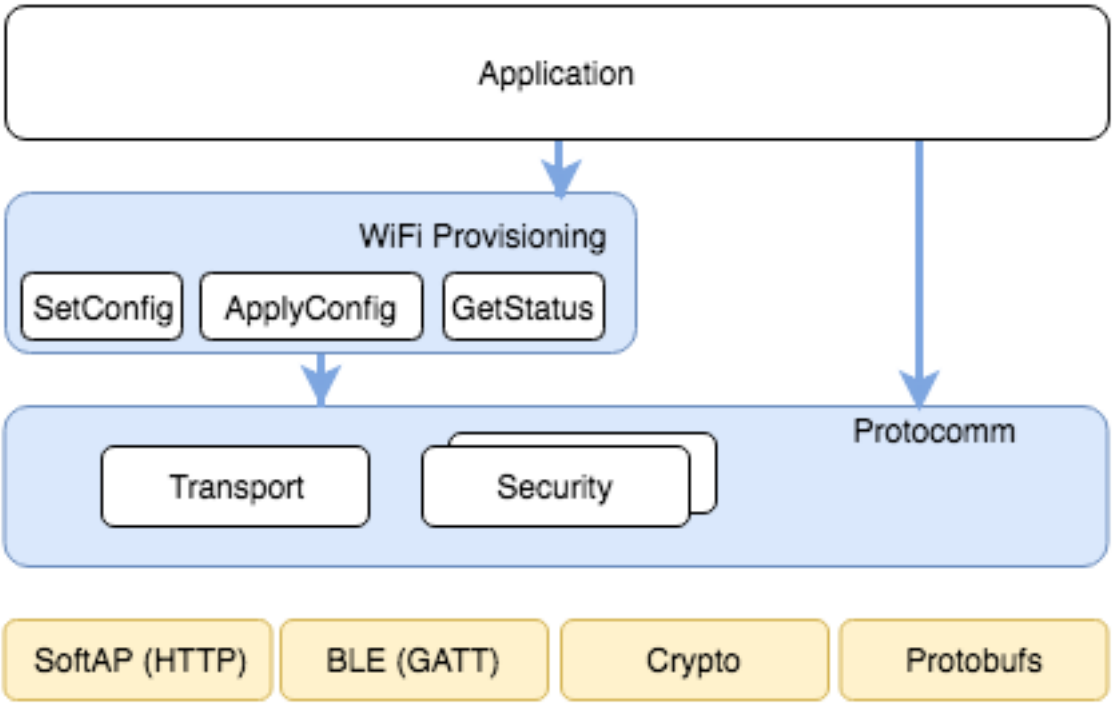


Fig. 29: Unified Provisioning Architecture

Security Schemes

At present, the unified provisioning supports the following security schemes:

1. Security 0

No security (No encryption).

2. Security 1

Curve25519-based key exchange, shared key derivation and AES256-CTR mode encryption of the data. It supports two modes :

a. Authorized - Proof of Possession (PoP) string used to authorize session and derive shared key.
b. No Auth (Null PoP) - Shared key derived through key exchange only.

3. Security 2

SRP6a-based shared key derivation and AES256-GCM mode encryption of the data.

Note: The respective security schemes need to be enabled through the project configuration menu. Please refer to *Enabling Protocomm Security Version* for more details.

Security 1 Scheme

The Security 1 scheme details are shown in the below sequence diagram:

Security 2 Scheme

The Security 2 scheme is based on the Secure Remote Password (SRP6a) protocol, see RFC 5054.

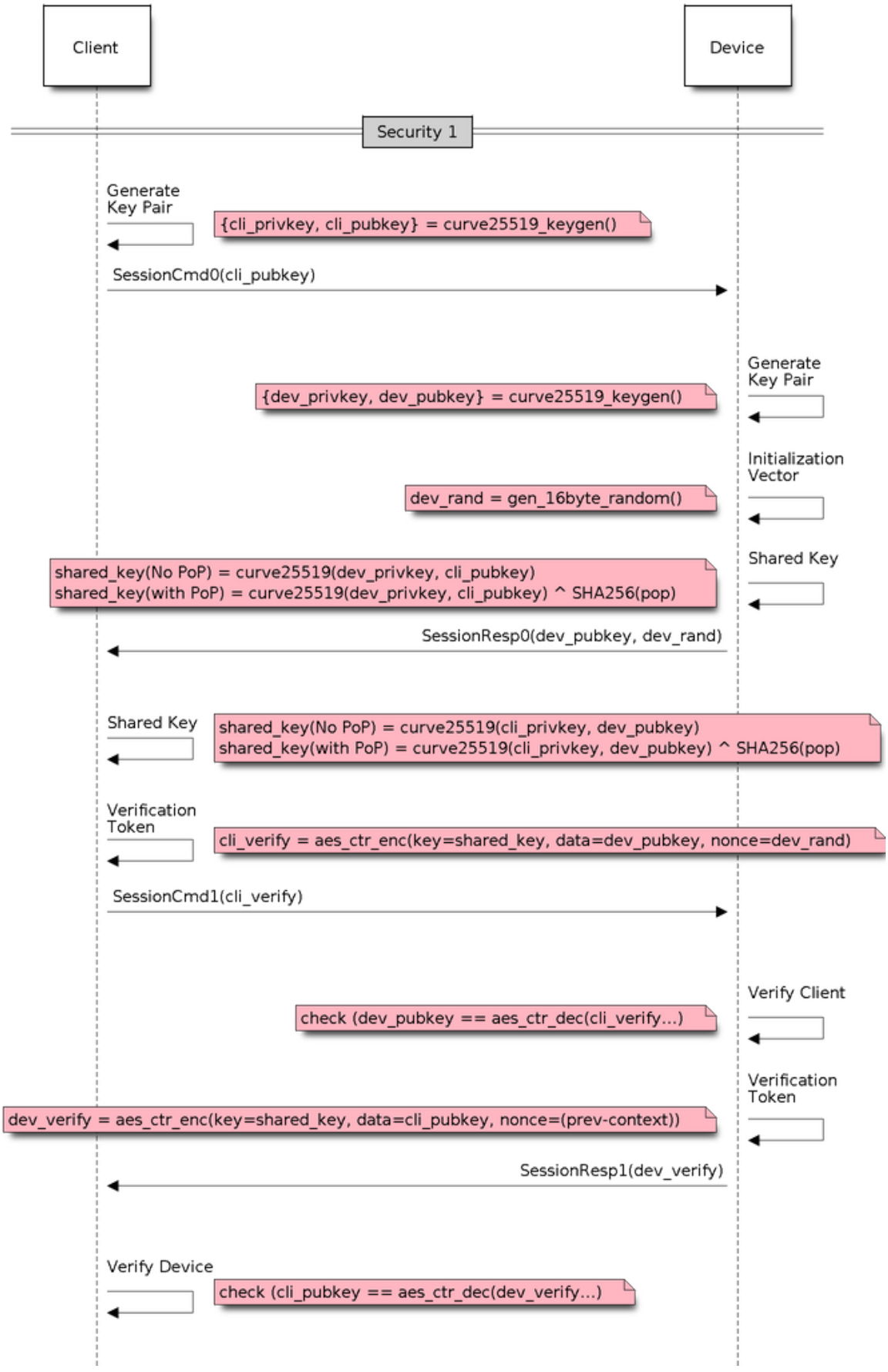


Fig. 30: Security 1

The protocol requires the Salt and Verifier to be generated beforehand with the help of the identifying username `I` and the plaintext password `p`. The Salt and Verifier are then stored on ESP32.

- The password `p` and the username `I` are to be provided to the Phone App (Provisioning entity) by suitable means, e.g., QR code sticker.

Details about the Security 2 scheme are shown in the below sequence diagram:

Sample Code

Please refer to *Protocol Communication* and *Wi-Fi Provisioning* for API guides and code snippets on example usage.

Application implementation can be found as an example under provisioning.

Provisioning Tools

Provisioning applications are available for various platforms, along with source code:

- **Android:**
  - Bluetooth LE Provisioning app on Play Store.
  - SoftAP Provisioning app on Play Store.
  - Source code on GitHub: esp-idf-provisioning-android.
- **iOS:**
  - Bluetooth LE Provisioning app on App Store.
  - SoftAP Provisioning app on App Store.
  - Source code on GitHub: esp-idf-provisioning-ios.
- Linux/macOS/Windows: tools/esp_prov, a Python-based command line tool for provisioning.

The phone applications offer simple UI and are thus more user centric, while the command-line application is useful as a debugging tool for developers.

2.8.3 Wi-Fi Provisioning

Overview

This component provides APIs that control the Wi-Fi provisioning service for receiving and configuring Wi-Fi credentials over SoftAP or Bluetooth LE transport via secure *Protocol Communication* sessions. The set of `wifi_prov_mgr_` APIs help quickly implement a provisioning service that has necessary features with minimal amount of code and sufficient flexibility.

Initialization *`wifi_prov_mgr_init()`* is called to configure and initialize the provisioning manager, and thus must be called prior to invoking any other `wifi_prov_mgr_` APIs. Note that the manager relies on other components of ESP-IDF, namely NVS, TCP/IP, Event Loop and Wi-Fi, and optionally mDNS, hence these components must be initialized beforehand. The manager can be de-initialized at any moment by making a call to *`wifi_prov_mgr_deinit()`*.

```
wifi_prov_mgr_config_t config = {
    .scheme = wifi_prov_scheme_ble,
    .scheme_event_handler = WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM
};

ESP_ERROR_CHECK( wifi_prov_mgr_init(config) );
```

The configuration structure *`wifi_prov_mgr_config_t`* has a few fields to specify the desired behavior of the manager:

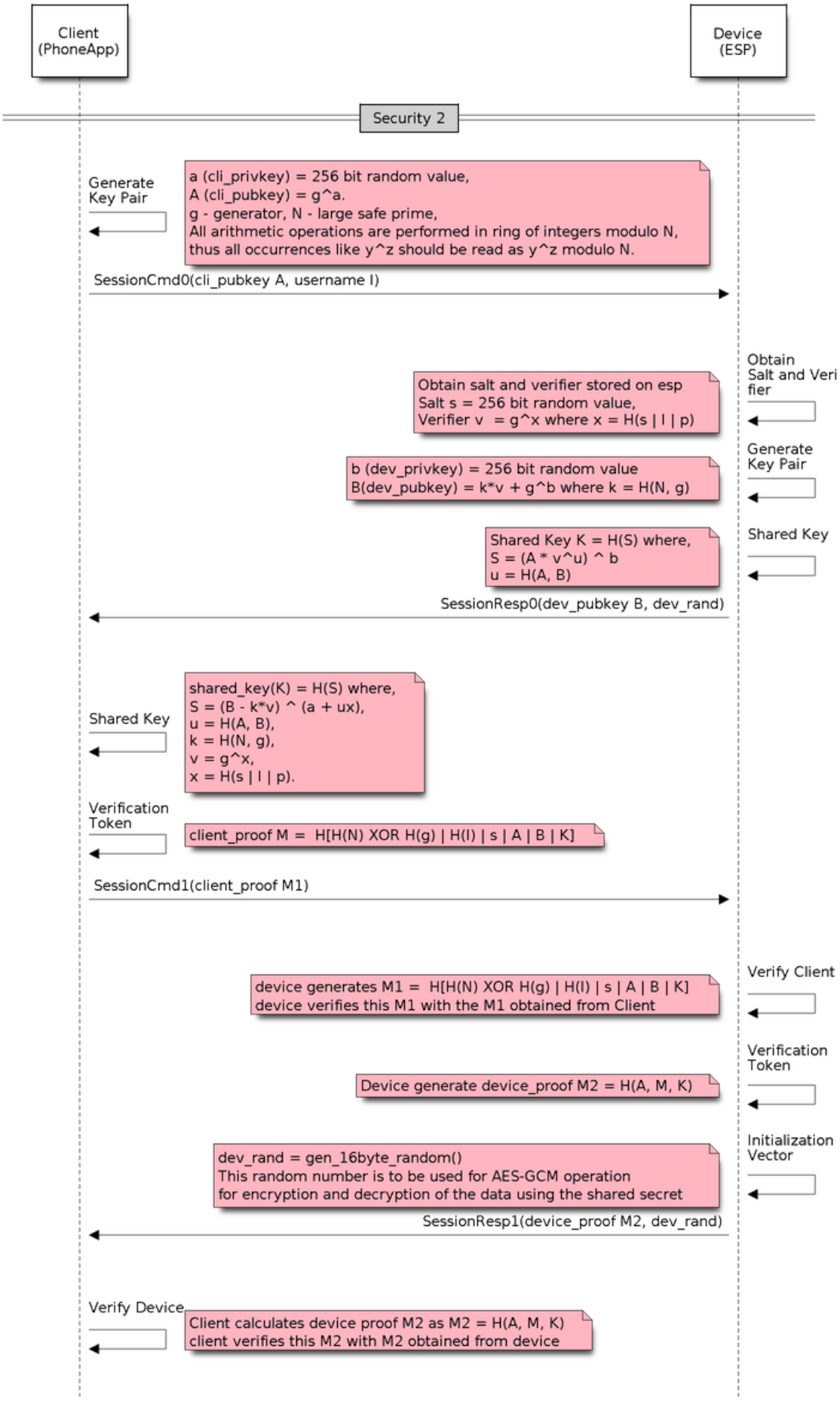


Fig. 31: Security 2

- *wifi_prov_mgr_config_t::scheme* - This is used to specify the provisioning scheme. Each scheme corresponds to one of the modes of transport supported by protocomm. Hence, support the following options:
  - wifi_prov_scheme_ble - Bluetooth LE transport and GATT Server for handling the provisioning commands.
  - wifi_prov_scheme_softap - Wi-Fi SoftAP transport and HTTP Server for handling the provisioning commands.
  - wifi_prov_scheme_console - Serial transport and console for handling the provisioning commands.
- *wifi_prov_mgr_config_t::scheme_event_handler*: An event handler defined along with the scheme. Choosing the appropriate scheme-specific event handler allows the manager to take care of certain matters automatically. Presently, this option is not used for either the SoftAP or Console-based provisioning, but is very convenient for Bluetooth LE. To understand how, we must recall that Bluetooth requires a substantial amount of memory to function, and once the provisioning is finished, the main application may want to reclaim back this memory (or part of it) if it needs to use either Bluetooth LE or classic Bluetooth. Also, upon every future reboot of a provisioned device, this reclamation of memory needs to be performed again. To reduce this complication in using wifi_prov_scheme_ble, the scheme-specific handlers have been defined, and depending upon the chosen handler, the Bluetooth LE/classic Bluetooth/BTDM memory is freed automatically when the provisioning manager is de-initialized. The available options are:
  - WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM - Free both classic Bluetooth and Bluetooth LE/BTDM memory. Used when the main application does not require Bluetooth at all.
  - WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE - Free only Bluetooth LE memory. Used when main application requires classic Bluetooth.
  - WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT - Free only classic Bluetooth. Used when main application requires Bluetooth LE. In this case freeing happens right when the manager is initialized.
  - WIFI_PROV_EVENT_HANDLER_NONE - Do not use any scheme specific handler. Used when the provisioning scheme is not Bluetooth LE, i.e., using SoftAP or Console, or when main application wants to handle the memory reclaiming on its own, or needs both Bluetooth LE and classic Bluetooth to function.
- *wifi_prov_mgr_config_t::app_event_handler* (Deprecated) - It is now recommended to catch WIFI_PROV_EVENT that is emitted to the default event loop handler. See definition of wifi_prov_cb_event_t for the list of events that are generated by the provisioning service. Here is an excerpt showing some of the provisioning events:

```
static void event_handler(void* arg, esp_event_base_t event_base,
                          int event_id, void* event_data)
{
    if (event_base == WIFI_PROV_EVENT) {
        switch (event_id) {
            case WIFI_PROV_START:
                ESP_LOGI(TAG, "Provisioning started");
                break;
            case WIFI_PROV_CRED_RECV: {
                wifi_sta_config_t *wifi_sta_cfg = (wifi_sta_config_t␣
↪*)event_data;
                ESP_LOGI(TAG, "Received Wi-Fi credentials"
                         "\n\tSSID     : %s\n\tPassword : %s",
                         (const char *) wifi_sta_cfg->ssid,
                         (const char *) wifi_sta_cfg->password);
                break;
            }
            case WIFI_PROV_CRED_FAIL: {
                wifi_prov_sta_fail_reason_t *reason = (wifi_prov_sta_fail_
↪reason_t *)event_data;
                ESP_LOGE(TAG, "Provisioning failed!\n\tReason : %s"
                         "\n\tPlease reset to factory and retry␣
↪provisioning",
```

(continues on next page)

(continued from previous page)

```
                        (*reason == WIFI_PROV_STA_AUTH_ERROR) ?
                        "Wi-Fi station authentication failed" : "Wi-Fi␣
↪access-point not found");
                break;
            }
            case WIFI_PROV_CRED_SUCCESS:
                ESP_LOGI(TAG, "Provisioning successful");
                break;
            case WIFI_PROV_END:
                /* De-initialize manager once provisioning is finished */
                wifi_prov_mgr_deinit();
                break;
            default:
                break;
        }
    }
}
```

The manager can be de-initialized at any moment by making a call to *wifi_prov_mgr_deinit()*.

Check the Provisioning State Whether the device is provisioned or not can be checked at runtime by calling *wifi_prov_mgr_is_provisioned()*. This internally checks if the Wi-Fi credentials are stored in NVS.

Note that presently the manager does not have its own NVS namespace for storage of Wi-Fi credentials, instead it relies on the `esp_wifi_` APIs to set and get the credentials stored in NVS from the default location.

If the provisioning state needs to be reset, any of the following approaches may be taken:

- The associated part of NVS partition has to be erased manually
- The main application must implement some logic to call `esp_wifi_` APIs for erasing the credentials at runtime
- The main application must implement some logic to force start the provisioning irrespective of the provisioning state

```
bool provisioned = false;
ESP_ERROR_CHECK( wifi_prov_mgr_is_provisioned(&provisioned) );
```

Start the Provisioning Service At the time of starting provisioning we need to specify a service name and the corresponding key, that is to say:

- A Wi-Fi SoftAP SSID and a passphrase, respectively, when the scheme is `wifi_prov_scheme_softap`.
- Bluetooth LE device name with the service key ignored when the scheme is `wifi_prov_scheme_ble`.

Also, since internally the manager uses `protocomm`, we have the option of choosing one of the security features provided by it:

- Security 1 is secure communication which consists of a prior handshake involving X25519 key exchange along with authentication using a proof of possession `pop`, followed by AES-CTR for encryption or decryption of subsequent messages.
- Security 0 is simply plain text communication. In this case the `pop` is simply ignored.

See *Unified Provisioning* for details about the security features.

```
const char *service_name = "my_device";
const char *service_key  = "password";

wifi_prov_security_t security = WIFI_PROV_SECURITY_1;
const char *pop = "abcd1234";

ESP_ERROR_CHECK( wifi_prov_mgr_start_provisioning(security, pop, service_
↪name, service_key) );
```

The provisioning service automatically finishes only if it receives valid Wi-Fi AP credentials followed by successful connection of device to the AP with IP obtained. Regardless of that, the provisioning service can be stopped at any moment by making a call to *wifi_prov_mgr_stop_provisioning()*.

Note: If the device fails to connect with the provided credentials, it does not accept new credentials anymore, but the provisioning service keeps on running, only to convey failure to the client, until the device is restarted. Upon restart, the provisioning state turns out to be true this time, as credentials are found in NVS, but the device does fail again to connect with those same credentials, unless an AP with the matching credentials somehow does become available. This situation can be fixed by resetting the credentials in NVS or force starting the provisioning service. This has been explained above in *Check the Provisioning State*.

Waiting for Completion Typically, the main application waits for the provisioning to finish, then de-initializes the manager to free up resources, and finally starts executing its own logic.

There are two ways for making this possible. The simpler way is to use a blocking call to *wifi_prov_mgr_wait()*.

```
// Start provisioning service
ESP_ERROR_CHECK( wifi_prov_mgr_start_provisioning(security, pop, service_
↪name, service_key) );

// Wait for service to complete
wifi_prov_mgr_wait();

// Finally de-initialize the manager
wifi_prov_mgr_deinit();
```

The other way is to use the default event loop handler to catch `WIFI_PROV_EVENT` and call *wifi_prov_mgr_deinit()* when event ID is `WIFI_PROV_END`:

```
static void event_handler(void* arg, esp_event_base_t event_base,
                          int event_id, void* event_data)
{
    if (event_base == WIFI_PROV_EVENT && event_id == WIFI_PROV_END) {
        /* De-initialize the manager once the provisioning is finished */
        wifi_prov_mgr_deinit();
    }
}
```

User Side Implementation When the service is started, the device to be provisioned is identified by the advertised service name, which, depending upon the selected transport, is either the Bluetooth LE device name or the SoftAP SSID.

When using SoftAP transport, for allowing service discovery, mDNS must be initialized before starting provisioning. In this case, the host name set by the main application is used, and the service type is internally set to `_esp_wifi_prov`.

When using Bluetooth LE transport, a custom 128-bit UUID should be set using *wifi_prov_scheme_ble_set_service_uuid()*. This UUID is to be included in the Bluetooth LE advertisement and corresponds to the primary GATT service that provides provisioning endpoints as GATT characteristics. Each GATT characteristic is formed using the primary service UUID as the base, with different auto-assigned 12th and 13th bytes, presumably counting from the 0th byte. Since an endpoint characteristic UUID is auto-assigned, it should not be used to identify the endpoint. Instead, client-side applications should identify the endpoints by reading the User Characteristic Description (`0x2901`) descriptor for each characteristic, which contains the endpoint name of the characteristic. For example, if the service UUID is set to `55cc035e-fb27-4f80-be02-3c60828b7451`, each endpoint characteristic is assigned a UUID like `55cc____-fb27-4f80-be02-3c60828b7451`, with unique values at the 12th and 13th bytes.

Once connected to the device, the provisioning-related protocomm endpoints can be identified as follows:

Table 9: Endpoints Provided by the Provisioning Service

Endpoint Name i.e., Bluetooth LE + GATT Server	URI, i.e., SoftAP + HTTP Server + mDNS	Description
prov-session	http://<mdns-hostname>.local/prov-session	Security endpoint used for session establishment
prov-scan	http://wifi-prov.local/prov-scan	the endpoint used for starting Wi-Fi scan and receiving scan results
prov-ctrl	http://wifi-prov.local/prov-ctrl	the endpoint used for controlling Wi-Fi provisioning state
prov-config	http://<mdns-hostname>.local/prov-config	the endpoint used for configuring Wi-Fi credentials on device
proto-ver	http://<mdns-hostname>.local/proto-ver	the endpoint for retrieving version info

Immediately after connecting, the client application may fetch the version/capabilities information from the `proto-ver` endpoint. All communications to this endpoint are unencrypted, hence necessary information, which may be relevant for deciding compatibility, can be retrieved before establishing a secure session. The response is in JSON format and looks like : `prov: { ver: v1.1, cap: [no_pop] }, my_app: { ver: 1.345, cap: [cloud, local_ctrl] },.....` Here label `prov` provides provisioning service version `ver` and capabilities `cap`. For now, only the `no_pop` capability is supported, which indicates that the service does not require proof of possession for authentication. Any application-related version or capabilities are given by other labels, e.g., `my_app` in this example. These additional fields are set using *`wifi_prov_mgr_set_app_info()`*.

User side applications need to implement the signature handshaking required for establishing and authenticating secure protocomm sessions as per the security scheme configured for use, which is not needed when the manager is configured to use protocomm security 0.

See *Unified Provisioning* for more details about the secure handshake and encryption used. Applications must use the `.proto` files found under protocomm/proto, which define the Protobuf message structures supported by `prov-session` endpoint.

Once a session is established, Wi-Fi credentials are configured using the following set of `wifi_config` commands, serialized as Protobuf messages with the corresponding `.proto` files that can be found under wifi_provisioning/proto:

- `get_status` - For querying the Wi-Fi connection status. The device responds with a status which is one of connecting, connected or disconnected. If the status is disconnected, a disconnection reason is also to be included in the status response.
- `set_config` - For setting the Wi-Fi connection credentials.
- `apply_config` - For applying the credentials saved during `set_config` and starting the Wi-Fi station.

After session establishment, the client can also request Wi-Fi scan results from the device. The results returned is a list of AP SSIDs, sorted in descending order of signal strength. This allows client applications to display APs nearby to the device at the time of provisioning, and users can select one of the SSIDs and provide the password which is then sent using the `wifi_config` commands described above. The `wifi_scan` endpoint supports the following protobuf commands :

- `scan_start` - For starting Wi-Fi scan with various options:
  - `blocking` (input) - If true, the command returns only when the scanning is finished.
  - `passive` (input) - If true, the scan is started in passive mode, which may be slower, instead of active mode.
  - `group_channels` (input) - This specifies whether to scan all channels in one go when zero, or perform scanning of channels in groups, with 120 ms delay between scanning of consecutive groups, and the value of this parameter sets the number of channels in each group. This is useful when transport mode is SoftAP, where scanning all channels in one go may not give the Wi-Fi driver enough time to send out beacons, and hence may cause disconnection with any connected stations. When scanning in groups, the manager waits for at least 120 ms after completing the scan on a group of channels, and thus allows the driver to send out the beacons. For example, given that the total number of Wi-Fi channels is 14, then setting `group_channels` to 3 creates 5 groups, with each group having 3 channels, except the last

one which has 14 % 3 = 2 channels. So, when the scan is started, the first 3 channels will be scanned, followed by a 120 ms delay, and then the next 3 channels, and so on, until all the 14 channels have been scanned.One may need to adjust this parameter as having only a few channels in a group may increase the overall scan time, while having too many may again cause disconnection. Usually, a value of 4 should work for most cases. Note that for any other mode of transport, e.g. Bluetooth LE, this can be safely set to 0, and hence achieve the shortest overall scanning time.
    - `period_ms` (input) - The scan parameter specifying how long to wait on each channel.
- `scan_status` - It gives the status of scanning process:
    - `scan_finished` (output) - When the scan has finished, this returns true.
    - `result_count` (output) - This gives the total number of results obtained till now. If the scan is yet happening, this number keeps on updating.
- `scan_result` - For fetching the scan results. This can be called even if the scan is still on going.
    - `start_index` (input) - Where the index starts from to fetch the entries from the results list.
    - `count` (input) - The number of entries to fetch from the starting index.
    - `entries` (output) - The list of entries returned. Each entry consists of `ssid`, `channel` and `rssi` information.

The client can also control the provisioning state of the device using `wifi_ctrl` endpoint. The `wifi_ctrl` endpoint supports the following protobuf commands:

- `ctrl_reset` - Resets internal state machine of the device and clears provisioned credentials only in case of provisioning failures.
- `ctrl_reprov` - Resets internal state machine of the device and clears provisioned credentials only in case the device is to be provisioned again for new credentials after a previous successful provisioning.

Additional Endpoints In case users want to have some additional protocomm endpoints customized to their requirements, this is done in two steps. First is creation of an endpoint with a specific name, and the second step is the registration of a handler for this endpoint. See *Protocol Communication* for the function signature of an endpoint handler. A custom endpoint must be created after initialization and before starting the provisioning service. Whereas, the protocomm handler is registered for this endpoint only after starting the provisioning service.

```
wifi_prov_mgr_init(config);
wifi_prov_mgr_endpoint_create("custom-endpoint");
wifi_prov_mgr_start_provisioning(security, pop, service_name, service_
↪key);
wifi_prov_mgr_endpoint_register("custom-endpoint", custom_ep_handler,␣
↪custom_ep_data);
```

When the provisioning service stops, the endpoint is unregistered automatically.

One can also choose to call *`wifi_prov_mgr_endpoint_unregister()`* to manually deactivate an endpoint at runtime. This can also be used to deactivate the internal endpoints used by the provisioning service.

When/How to Stop the Provisioning Service? The default behavior is that once the device successfully connects using the Wi-Fi credentials set by the `apply_config` command, the provisioning service stops, and Bluetooth LE or SoftAP turns off, automatically after responding to the next `get_status` command. If `get_status` command is not received by the device, the service stops after a 30s timeout.

On the other hand, if device is not able to connect using the provided Wi-Fi credentials, due to incorrect SSID or passphrase, the service keeps running, and `get_status` keeps responding with disconnected status and reason for disconnection. Any further attempts to provide another set of Wi-Fi credentials, are to be rejected. These credentials are preserved, unless the provisioning service is force started, or NVS erased.

If this default behavior is not desired, it can be disabled by calling *`wifi_prov_mgr_disable_auto_stop()`*. Now the provisioning service stops only after an explicit call to *`wifi_prov_mgr_stop_provisioning()`*, which returns immediately after scheduling a task for stopping the service. The service stops after a certain delay and `WIFI_PROV_END` event gets emitted. This delay is specified by the argument to *`wifi_prov_mgr_disable_auto_stop()`*.

The customized behavior is useful for applications which want the provisioning service to be stopped some time after the Wi-Fi connection is successfully established. For example, if the application requires the device to connect to some cloud service and obtain another set of credentials, and exchange these credentials over a custom protocomm endpoint, then after successfully doing so, stop the provisioning service by calling `wifi_prov_mgr_stop_provisioning()` inside the protocomm handler itself. The right amount of delay ensures that the transport resources are freed only after the response from the protocomm handler reaches the client side application.

Application Examples

For complete example implementation see provisioning/wifi_prov_mgr.

Provisioning Tools

Provisioning applications are available for various platforms, along with source code:

- **Android:**
  - Bluetooth LE Provisioning app on Play Store.
  - SoftAP Provisioning app on Play Store.
  - Source code on GitHub: esp-idf-provisioning-android.
- **iOS:**
  - Bluetooth LE Provisioning app on App Store.
  - SoftAP Provisioning app on App Store.
  - Source code on GitHub: esp-idf-provisioning-ios.
- Linux/MacOS/Windows: tools/esp_prov, a Python-based command-line tool for provisioning.

The phone applications offer simple UI and are thus more user centric, while the command-line application is useful as a debugging tool for developers.

API Reference

Header File

- components/wifi_provisioning/include/wifi_provisioning/manager.h

Functions

esp_err_t **wifi_prov_mgr_init** (*wifi_prov_mgr_config_t* config)

Initialize provisioning manager instance.

Configures the manager and allocates internal resources

Configuration specifies the provisioning scheme (transport) and event handlers

Event WIFI_PROV_INIT is emitted right after initialization is complete

Parameters **config** –**[in]** Configuration structure

Returns

- ESP_OK : Success
- ESP_FAIL : Fail

void **wifi_prov_mgr_deinit** (void)

Stop provisioning (if running) and release resource used by the manager.

Event WIFI_PROV_DEINIT is emitted right after de-initialization is finished

If provisioning service is still active when this API is called, it first stops the service, hence emitting WIFI_PROV_END, and then performs the de-initialization

esp_err_t **wifi_prov_mgr_is_provisioned**(bool *provisioned)

Checks if device is provisioned.

This checks if Wi-Fi credentials are present on the NVS

The Wi-Fi credentials are assumed to be kept in the same NVS namespace as used by esp_wifi component

If one were to call esp_wifi_set_config() directly instead of going through the provisioning process, this function will still yield true (i.e. device will be found to be provisioned)

Note: Calling wifi_prov_mgr_start_provisioning() automatically resets the provision state, irrespective of what the state was prior to making the call.

Parameters **provisioned** –**[out]** True if provisioned, else false

Returns

- ESP_OK : Retrieved provision state successfully
- ESP_FAIL : Wi-Fi not initialized
- ESP_ERR_INVALID_ARG : Null argument supplied

esp_err_t **wifi_prov_mgr_start_provisioning**(*wifi_prov_security_t* security, const void *wifi_prov_sec_params, const char *service_name, const char *service_key)

Start provisioning service.

This starts the provisioning service according to the scheme configured at the time of initialization. For scheme :

- wifi_prov_scheme_ble : This starts protocomm_ble, which internally initializes BLE transport and starts GATT server for handling provisioning requests
- wifi_prov_scheme_softap : This activates SoftAP mode of Wi-Fi and starts protocomm_httpd, which internally starts an HTTP server for handling provisioning requests (If mDNS is active it also starts advertising service with type _esp_wifi_prov._tcp)

Event WIFI_PROV_START is emitted right after provisioning starts without failure

---

Note: This API will start provisioning service even if device is found to be already provisioned, i.e. wifi_prov_mgr_is_provisioned() yields true

---

Parameters

- **security** –**[in]** Specify which protocomm security scheme to use :
  - WIFI_PROV_SECURITY_0 : For no security
  - WIFI_PROV_SECURITY_1 : x25519 secure handshake for session establishment followed by AES-CTR encryption of provisioning messages
  - WIFI_PROV_SECURITY_2: SRP6a based authentication and key exchange followed by AES-GCM encryption/decryption of provisioning messages
- **wifi_prov_sec_params** –**[in]** Pointer to security params (NULL if not needed). This is not needed for protocomm security 0 This pointer should hold the struct of type wifi_prov_security1_params_t for protocomm security 1 and wifi_prov_security2_params_t for protocomm security 2 respectively. This pointer and its contents should be valid till the provisioning service is running and has not been stopped or de-inited.
- **service_name** –**[in]** Unique name of the service. This translates to:
  - Wi-Fi SSID when provisioning mode is softAP
  - Device name when provisioning mode is BLE
- **service_key** –**[in]** Key required by client to access the service (NULL if not needed). This translates to:
  - Wi-Fi password when provisioning mode is softAP
  - ignored when provisioning mode is BLE

Returns

- ESP_OK : Provisioning started successfully
- ESP_FAIL : Failed to start provisioning service
- ESP_ERR_INVALID_STATE : Provisioning manager not initialized or already started

void **wifi_prov_mgr_stop_provisioning**(void)

Stop provisioning service.

If provisioning service is active, this API will initiate a process to stop the service and return. Once the service actually stops, the event WIFI_PROV_END will be emitted.

If wifi_prov_mgr_deinit() is called without calling this API first, it will automatically stop the provisioning service and emit the WIFI_PROV_END, followed by WIFI_PROV_DEINIT, before returning.

This API will generally be used along with wifi_prov_mgr_disable_auto_stop() in the scenario when the main application has registered its own endpoints, and wishes that the provisioning service is stopped only when some protocomm command from the client side application is received.

Calling this API inside an endpoint handler, with sufficient cleanup_delay, will allow the response / acknowledgment to be sent successfully before the underlying protocomm service is stopped.

Cleaup_delay is set when calling wifi_prov_mgr_disable_auto_stop(). If not specified, it defaults to 1000ms.

For straightforward cases, using this API is usually not necessary as provisioning is stopped automatically once WIFI_PROV_CRED_SUCCESS is emitted. Stopping is delayed (maximum 30 seconds) thus allowing the client side application to query for Wi-Fi state, i.e. after receiving the first query and sending `Wi-Fi state connected` response the service is stopped immediately.

void **wifi_prov_mgr_wait**(void)

Wait for provisioning service to finish.

Calling this API will block until provisioning service is stopped i.e. till event WIFI_PROV_END is emitted.

This will not block if provisioning is not started or not initialized.

esp_err_t **wifi_prov_mgr_disable_auto_stop**(uint32_t cleanup_delay)

Disable auto stopping of provisioning service upon completion.

By default, once provisioning is complete, the provisioning service is automatically stopped, and all endpoints (along with those registered by main application) are deactivated.

This API is useful in the case when main application wishes to close provisioning service only after it receives some protocomm command from the client side app. For example, after connecting to Wi-Fi, the device may want to connect to the cloud, and only once that is successfully, the device is said to be fully configured. But, then it is upto the main application to explicitly call wifi_prov_mgr_stop_provisioning() later when the device is fully configured and the provisioning service is no longer required.

Note: This must be called before executing wifi_prov_mgr_start_provisioning()

Parameters **cleanup_delay** –**[in]** Sets the delay after which the actual cleanup of transport related resources is done after a call to wifi_prov_mgr_stop_provisioning() returns. Minimum allowed value is 100ms. If not specified, this will default to 1000ms.

Returns

- ESP_OK : Success
- ESP_ERR_INVALID_STATE : Manager not initialized or provisioning service already started

esp_err_t **wifi_prov_mgr_set_app_info**(const char *label, const char *version, const char **capabilities, size_t total_capabilities)

Set application version and capabilities in the JSON data returned by proto-ver endpoint.

This function can be called multiple times, to specify information about the various application specific services running on the device, identified by unique labels.

The provisioning service itself registers an entry in the JSON data, by the label "prov", containing only provisioning service version and capabilities. Application services should use a label other than "prov" so as not to overwrite this.

Note: This must be called before executing wifi_prov_mgr_start_provisioning()

Parameters
- **`label`** –**[in]** String indicating the application name.
- **`version`** –**[in]** String indicating the application version. There is no constraint on format.
- **`capabilities`** –**[in]** Array of strings with capabilities. These could be used by the client side app to know the application registered endpoint capabilities
- **`total_capabilities`** –**[in]** Size of capabilities array

Returns
- ESP_OK : Success
- ESP_ERR_INVALID_STATE : Manager not initialized or provisioning service already started
- ESP_ERR_NO_MEM : Failed to allocate memory for version string
- ESP_ERR_INVALID_ARG : Null argument

esp_err_t **`wifi_prov_mgr_endpoint_create`**(const char *ep_name)

Create an additional endpoint and allocate internal resources for it.

This API is to be called by the application if it wants to create an additional endpoint. All additional endpoints will be assigned UUIDs starting from 0xFF54 and so on in the order of execution.

protocomm handler for the created endpoint is to be registered later using wifi_prov_mgr_endpoint_register() after provisioning has started.

Note: This API can only be called BEFORE provisioning is started

Note: Additional endpoints can be used for configuring client provided parameters other than Wi-Fi credentials, that are necessary for the main application and hence must be set prior to starting the application

Note: After session establishment, the additional endpoints must be targeted first by the client side application before sending Wi-Fi configuration, because once Wi-Fi configuration finishes the provisioning service is stopped and hence all endpoints are unregistered

Parameters **`ep_name`** –**[in]** unique name of the endpoint

Returns
- ESP_OK : Success
- ESP_FAIL : Failure

esp_err_t **`wifi_prov_mgr_endpoint_register`**(const char *ep_name, *protocomm_req_handler_t* handler, void *user_ctx)

Register a handler for the previously created endpoint.

This API can be called by the application to register a protocomm handler to any endpoint that was created using wifi_prov_mgr_endpoint_create().

Note: This API can only be called AFTER provisioning has started

Note: Additional endpoints can be used for configuring client provided parameters other than Wi-Fi credentials, that are necessary for the main application and hence must be set prior to starting the application

Note: After session establishment, the additional endpoints must be targeted first by the client side application before sending Wi-Fi configuration, because once Wi-Fi configuration finishes the provisioning service is stopped and hence all endpoints are unregistered

Parameters
- **`ep_name`** –**[in]** Name of the endpoint
- **`handler`** –**[in]** Endpoint handler function
- **`user_ctx`** –**[in]** User data

Returns
- ESP_OK : Success
- ESP_FAIL : Failure

void **`wifi_prov_mgr_endpoint_unregister`**(const char *ep_name)

Unregister the handler for an endpoint.

This API can be called if the application wants to selectively unregister the handler of an endpoint while the provisioning is still in progress.

All the endpoint handlers are unregistered automatically when the provisioning stops.

Parameters **`ep_name`** –**[in]** Name of the endpoint

esp_err_t **`wifi_prov_mgr_get_wifi_state`**(*wifi_prov_sta_state_t* *state)

Get state of Wi-Fi Station during provisioning.

Parameters **`state`** –**[out]** Pointer to wifi_prov_sta_state_t variable to be filled

Returns
- ESP_OK : Successfully retrieved Wi-Fi state
- ESP_FAIL : Provisioning app not running

esp_err_t **`wifi_prov_mgr_get_wifi_disconnect_reason`**(*wifi_prov_sta_fail_reason_t* *reason)

Get reason code in case of Wi-Fi station disconnection during provisioning.

Parameters **`reason`** –**[out]** Pointer to wifi_prov_sta_fail_reason_t variable to be filled

Returns
- ESP_OK : Successfully retrieved Wi-Fi disconnect reason
- ESP_FAIL : Provisioning app not running

esp_err_t **`wifi_prov_mgr_configure_sta`**(*wifi_config_t* *wifi_cfg)

Runs Wi-Fi as Station with the supplied configuration.

Configures the Wi-Fi station mode to connect to the AP with SSID and password specified in config structure and sets Wi-Fi to run as station.

This is automatically called by provisioning service upon receiving new credentials.

If credentials are to be supplied to the manager via a different mode other than through protocomm, then this API needs to be called.

Event WIFI_PROV_CRED_RECV is emitted after credentials have been applied and Wi-Fi station started

Parameters **`wifi_cfg`** –**[in]** Pointer to Wi-Fi configuration structure

Returns
- ESP_OK : Wi-Fi configured and started successfully

- ESP_FAIL : Failed to set configuration

esp_err_t **wifi_prov_mgr_reset_provisioning**(void)

Reset Wi-Fi provisioning config.

Calling this API will restore WiFi stack persistent settings to default values.

Returns
- ESP_OK : Reset provisioning config successfully
- ESP_FAIL : Failed to reset provisioning config

esp_err_t **wifi_prov_mgr_reset_sm_state_on_failure**(void)

Reset internal state machine and clear provisioned credentials.

This API can be used to restart provisioning in case invalid credentials are entered.

Returns
- ESP_OK : Reset provisioning state machine successfully
- ESP_FAIL : Failed to reset provisioning state machine
- ESP_ERR_INVALID_STATE : Manager not initialized

Structures

struct **wifi_prov_event_handler_t**

Event handler that is used by the manager while provisioning service is active.

Public Members

wifi_prov_cb_func_t **event_cb**

Callback function to be executed on provisioning events

void ***user_data**

User context data to pass as parameter to callback function

struct **wifi_prov_scheme**

Structure for specifying the provisioning scheme to be followed by the manager.

Note: Ready to use schemes are available:

- wifi_prov_scheme_ble : for provisioning over BLE transport + GATT server
- wifi_prov_scheme_softap : for provisioning over SoftAP transport + HTTP server
- wifi_prov_scheme_console : for provisioning over Serial UART transport + Console (for debugging)

Public Members

esp_err_t (***prov_start**)(*protocomm_t* *pc, void *config)

Function which is to be called by the manager when it is to start the provisioning service associated with a protocomm instance and a scheme specific configuration

esp_err_t (***prov_stop**)(*protocomm_t* *pc)

Function which is to be called by the manager to stop the provisioning service previously associated with a protocomm instance

void *(***new_config**)(void)

Function which is to be called by the manager to generate a new configuration for the provisioning service, that is to be passed to *prov_start()*

void (***delete_config**)(void *config)

Function which is to be called by the manager to delete a configuration generated using *new_config()*

esp_err_t (***set_config_service**)(void *config, const char *service_name, const char *service_key)

Function which is to be called by the manager to set the service name and key values in the configuration structure

esp_err_t (***set_config_endpoint**)(void *config, const char *endpoint_name, uint16_t uuid)

Function which is to be called by the manager to set a protocomm endpoint with an identifying name and UUID in the configuration structure

wifi_mode_t **wifi_mode**

Sets mode of operation of Wi-Fi during provisioning This is set to :

- WIFI_MODE_APSTA for SoftAP transport
- WIFI_MODE_STA for BLE transport

struct **wifi_prov_mgr_config_t**

Structure for specifying the manager configuration.

Public Members

wifi_prov_scheme_t **scheme**

Provisioning scheme to use. Following schemes are already available:

- wifi_prov_scheme_ble : for provisioning over BLE transport + GATT server
- wifi_prov_scheme_softap : for provisioning over SoftAP transport + HTTP server + mDNS (optional)
- wifi_prov_scheme_console : for provisioning over Serial UART transport + Console (for debugging)

wifi_prov_event_handler_t **scheme_event_handler**

Event handler required by the scheme for incorporating scheme specific behavior while provisioning manager is running. Various options may be provided by the scheme for setting this field. Use WIFI_PROV_EVENT_HANDLER_NONE when not used. When using scheme wifi_prov_scheme_ble, the following options are available:

- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM
- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE
- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT

wifi_prov_event_handler_t **app_event_handler**

Event handler that can be set for the purpose of incorporating application specific behavior. Use WIFI_PROV_EVENT_HANDLER_NONE when not used.

Macros

WIFI_PROV_EVENT_HANDLER_NONE

Event handler can be set to none if not used.

Type Definitions

typedef void (*`wifi_prov_cb_func_t`)(void *user_data, *wifi_prov_cb_event_t* event, void *event_data)

typedef struct *wifi_prov_scheme* `wifi_prov_scheme_t`

Structure for specifying the provisioning scheme to be followed by the manager.

Note: Ready to use schemes are available:

- wifi_prov_scheme_ble : for provisioning over BLE transport + GATT server
- wifi_prov_scheme_softap : for provisioning over SoftAP transport + HTTP server
- wifi_prov_scheme_console : for provisioning over Serial UART transport + Console (for debugging)

---

typedef enum *wifi_prov_security* `wifi_prov_security_t`

Security modes supported by the Provisioning Manager.

These are same as the security modes provided by protocomm

typedef *protocomm_security2_params_t* `wifi_prov_security2_params_t`

Security 2 params structure This needs to be passed when using WIFI_PROV_SECURITY_2.

Enumerations

enum `wifi_prov_cb_event_t`

Events generated by manager.

These events are generated in order of declaration and, for the stretch of time between initialization and de-initialization of the manager, each event is signaled only once

Values:

enumerator `WIFI_PROV_INIT`

Emitted when the manager is initialized

enumerator `WIFI_PROV_START`

Indicates that provisioning has started

enumerator `WIFI_PROV_CRED_RECV`

Emitted when Wi-Fi AP credentials are received via `protocomm` endpoint `wifi_config`. The event data in this case is a pointer to the corresponding *`wifi_sta_config_t`* structure

enumerator `WIFI_PROV_CRED_FAIL`

Emitted when device fails to connect to the AP of which the credentials were received earlier on event `WIFI_PROV_CRED_RECV`. The event data in this case is a pointer to the disconnection reason code with type `wifi_prov_sta_fail_reason_t`

enumerator `WIFI_PROV_CRED_SUCCESS`

Emitted when device successfully connects to the AP of which the credentials were received earlier on event `WIFI_PROV_CRED_RECV`

enumerator `WIFI_PROV_END`

Signals that provisioning service has stopped

enumerator **WIFI_PROV_DEINIT**

Signals that manager has been de-initialized

enum **wifi_prov_security**

Security modes supported by the Provisioning Manager.

These are same as the security modes provided by protocomm

Values:

enumerator **WIFI_PROV_SECURITY_0**

No security (plain-text communication)

enumerator **WIFI_PROV_SECURITY_1**

This secure communication mode consists of X25519 key exchange

- proof of possession (pop) based authentication
- AES-CTR encryption

enumerator **WIFI_PROV_SECURITY_2**

This secure communication mode consists of SRP6a based authentication and key exchange

- AES-GCM encryption/decryption

Header File

- components/wifi_provisioning/include/wifi_provisioning/scheme_ble.h

Functions

void **wifi_prov_scheme_ble_event_cb_free_btdm**(void *user_data, *wifi_prov_cb_event_t* event, void *event_data)

void **wifi_prov_scheme_ble_event_cb_free_ble**(void *user_data, *wifi_prov_cb_event_t* event, void *event_data)

void **wifi_prov_scheme_ble_event_cb_free_bt**(void *user_data, *wifi_prov_cb_event_t* event, void *event_data)

esp_err_t **wifi_prov_scheme_ble_set_service_uuid**(uint8_t *uuid128)

Set the 128 bit GATT service UUID used for provisioning.

This API is used to override the default 128 bit provisioning service UUID, which is 0000ffff-0000-1000-8000-00805f9b34fb.

This must be called before starting provisioning, i.e. before making a call to wifi_prov_mgr_start_provisioning(), otherwise the default UUID will be used.

Note: The data being pointed to by the argument must be valid atleast till provisioning is started. Upon start, the manager will store an internal copy of this UUID, and this data can be freed or invalidated afterwords.

Parameters **uuid128** – **[in]** A custom 128 bit UUID

Returns

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null argument

esp_err_t **wifi_prov_scheme_ble_set_mfg_data**(uint8_t *mfg_data, ssize_t mfg_data_len)

Set manufacturer specific data in scan response.

This must be called before starting provisioning, i.e. before making a call to wifi_prov_mgr_start_provisioning().

Note: It is important to understand that length of custom manufacturer data should be within limits. The manufacturer data goes into scan response along with BLE device name. By default, BLE device name length is of 11 Bytes, however it can vary as per application use case. So, one has to honour the scan response data size limits i.e. (mfg_data_len + 2) < 31 - (device_name_length + 2). If the mfg_data length exceeds this limit, the length will be truncated.

Parameters
- **mfg_data** –**[in]** Custom manufacturer data
- **mfg_data_len** –**[in]** Manufacturer data length

Returns
- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null argument

Macros

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT

Header File

- components/wifi_provisioning/include/wifi_provisioning/scheme_softap.h

Functions

void **wifi_prov_scheme_softap_set_httpd_handle**(void *handle)

Provide HTTPD Server handle externally.

Useful in cases wherein applications need the webserver for some different operations, and do not want the wifi provisioning component to start/stop a new instance.

Note: This API should be called before wifi_prov_mgr_start_provisioning()

Parameters **handle** –**[in]** Handle to HTTPD server instance

Header File

- components/wifi_provisioning/include/wifi_provisioning/scheme_console.h

Header File

- components/wifi_provisioning/include/wifi_provisioning/wifi_config.h

Functions

esp_err_t **wifi_prov_config_data_handler**(uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen, void *priv_data)

Handler for receiving and responding to requests from master.

This is to be registered as the `wifi_config` endpoint handler (protocomm `protocomm_req_handler_t`) using `protocomm_add_endpoint()`

Structures

struct **wifi_prov_sta_conn_info_t**

WiFi STA connected status information.

Public Members

char **ip_addr**[IP4ADDR_STRLEN_MAX]

IP Address received by station

char **bssid**[6]

BSSID of the AP to which connection was estalished

char **ssid**[33]

SSID of the to which connection was estalished

uint8_t **channel**

Channel of the AP

uint8_t **auth_mode**

Authorization mode of the AP

struct **wifi_prov_config_get_data_t**

WiFi status data to be sent in response to `get_status` request from master.

Public Members

wifi_prov_sta_state_t **wifi_state**

WiFi state of the station

wifi_prov_sta_fail_reason_t **fail_reason**

Reason for disconnection (valid only when `wifi_state` is `WIFI_STATION_DISCONNECTED`)

wifi_prov_sta_conn_info_t **conn_info**

Connection information (valid only when `wifi_state` is `WIFI_STATION_CONNECTED`)

struct **wifi_prov_config_set_data_t**

WiFi config data received by slave during `set_config` request from master.

Public Members

char **ssid**[33]

SSID of the AP to which the slave is to be connected

char **password**[64]

Password of the AP

char **bssid**[6]

BSSID of the AP

uint8_t **channel**

Channel of the AP

struct **wifi_prov_config_handlers**

Internal handlers for receiving and responding to protocomm requests from master.

This is to be passed as priv_data for protocomm request handler (refer to `wifi_prov_config_data_handler()`) when calling `protocomm_add_endpoint()`.

Public Members

esp_err_t (***get_status_handler**)(*wifi_prov_config_get_data_t* *resp_data, *wifi_prov_ctx_t* **ctx)

Handler function called when connection status of the slave (in WiFi station mode) is requested

esp_err_t (***set_config_handler**)(const *wifi_prov_config_set_data_t* *req_data, *wifi_prov_ctx_t* **ctx)

Handler function called when WiFi connection configuration (eg. AP SSID, password, etc.) of the slave (in WiFi station mode) is to be set to user provided values

esp_err_t (***apply_config_handler**)(*wifi_prov_ctx_t* **ctx)

Handler function for applying the configuration that was set in `set_config_handler`. After applying the station may get connected to the AP or may fail to connect. The slave must be ready to convey the updated connection status information when `get_status_handler` is invoked again by the master.

wifi_prov_ctx_t ***ctx**

Context pointer to be passed to above handler functions upon invocation

Type Definitions

typedef struct wifi_prov_ctx **wifi_prov_ctx_t**

Type of context data passed to each get/set/apply handler function set in *`wifi_prov_config_handlers`* structure.

This is passed as an opaque pointer, thereby allowing it be defined later in application code as per requirements.

typedef struct *wifi_prov_config_handlers* **wifi_prov_config_handlers_t**

Internal handlers for receiving and responding to protocomm requests from master.

This is to be passed as priv_data for protocomm request handler (refer to `wifi_prov_config_data_handler()`) when calling `protocomm_add_endpoint()`.

Enumerations

enum **wifi_prov_sta_state_t**

WiFi STA status for conveying back to the provisioning master.

Values:

enumerator **WIFI_PROV_STA_CONNECTING**

enumerator **WIFI_PROV_STA_CONNECTED**

enumerator **WIFI_PROV_STA_DISCONNECTED**

enum **wifi_prov_sta_fail_reason_t**

WiFi STA connection fail reason.

Values:

enumerator **WIFI_PROV_STA_AUTH_ERROR**

enumerator **WIFI_PROV_STA_AP_NOT_FOUND**

Code examples for above API are provided in the provisioning directory of ESP-IDF examples.

Code example for above API is provided in wifi/smart_config.

Code example for above API is provided in wifi/wifi_easy_connect/dpp-enrollee.

2.9 Storage API

2.9.1 FAT Filesystem Support

ESP-IDF uses the FatFs library to work with FAT filesystems. FatFs resides in the `fatfs` component. Although the library can be used directly, many of its features can be accessed via VFS using the C standard library and POSIX API functions.

Additionally, FatFs has been modified to support the runtime pluggable disk I/O layer. This allows mapping of FatFs drives to physical disks at runtime.

Using FatFs with VFS

The header file fatfs/vfs/esp_vfs_fat.h defines the functions for connecting FatFs and VFS.

The function *`esp_vfs_fat_register()`* allocates a `FATFS` structure and registers a given path prefix in VFS. Subsequent operations on files starting with this prefix are forwarded to FatFs APIs.

The function *`esp_vfs_fat_unregister_path()`* deletes the registration with VFS, and frees the `FATFS` structure.

Most applications use the following workflow when working with `esp_vfs_fat_` functions:

1. **Call *`esp_vfs_fat_register()`* to specify:**
   - Path prefix where to mount the filesystem (e.g., `"/sdcard"`, `"/spiflash"`)
   - FatFs drive number

- A variable which will receive the pointer to the `FATFS` structure

2. Call *`ff_diskio_register()`* to register the disk I/O driver for the drive number used in Step 1.
3. Call the FatFs function `f_mount`, and optionally `f_fdisk`, `f_mkfs`, to mount the filesystem using the same drive number which was passed to *`esp_vfs_fat_register()`*. For more information, see FatFs documentation.
4. Call the C standard library and POSIX API functions to perform such actions on files as open, read, write, erase, copy, etc. Use paths starting with the path prefix passed to *`esp_vfs_register()`* (for example, `"/sdcard/hello.txt"`). The filesystem uses 8.3 filenames format (SFN) by default. If you need to use long filenames (LFN), enable the *CONFIG_FATFS_LONG_FILENAMES* option. More details on the FatFs filenames are available here.
5. Optionally, by enabling the option *CONFIG_FATFS_USE_FASTSEEK*, you can use the POSIX lseek function to perform it faster. The fast seek will not work for files in write mode, so to take advantage of fast seek, you should open (or close and then reopen) the file in read-only mode.
6. Optionally, call the FatFs library functions directly. In this case, use paths without a VFS prefix (for example, `"/hello.txt"`).
7. Close all open files.
8. Call the FatFs function `f_mount` for the same drive number with NULL `FATFS*` argument to unmount the filesystem.
9. Call the FatFs function *`ff_diskio_register()`* with NULL `ff_diskio_impl_t*` argument and the same drive number to unregister the disk I/O driver.
10. Call *`esp_vfs_fat_unregister_path()`* with the path where the file system is mounted to remove FatFs from VFS, and free the `FATFS` structure allocated in Step 1.

The convenience functions *`esp_vfs_fat_sdmmc_mount()`*, *`esp_vfs_fat_sdspi_mount()`*, and *`esp_vfs_fat_sdcard_unmount()`* wrap the steps described above and also handle SD card initialization. These functions are described in the next section.

Using FatFs with VFS and SD Cards

The header file fatfs/vfs/esp_vfs_fat.h defines convenience functions *`esp_vfs_fat_sdmmc_mount()`*, *`esp_vfs_fat_sdspi_mount()`*, and *`esp_vfs_fat_sdcard_unmount()`*. These functions perform Steps 1–3 and 7–9 respectively and handle SD card initialization, but provide only limited error handling. Developers are encouraged to check its source code and incorporate more advanced features into production applications.

The convenience function *`esp_vfs_fat_sdmmc_unmount()`* unmounts the filesystem and releases the resources acquired by *`esp_vfs_fat_sdmmc_mount()`*.

Using FatFs with VFS in Read-Only Mode

The header file fatfs/vfs/esp_vfs_fat.h also defines the convenience functions *`esp_vfs_fat_spiflash_mount_ro()`* and *`esp_vfs_fat_spiflash_unmount_ro()`*. These functions perform Steps 1-3 and 7-9 respectively for read-only FAT partitions. These are particularly helpful for data partitions written only once during factory provisioning which will not be changed by production application throughout the lifetime of the hardware.

FatFS Disk IO Layer

FatFs has been extended with API functions that register the disk I/O driver at runtime.

These APIs provide implementation of disk I/O functions for SD/MMC cards and can be registered for the given FatFs drive number using the function *`ff_diskio_register_sdmmc()`*.

void **`ff_diskio_register`**(BYTE pdrv, const *ff_diskio_impl_t* *discio_impl)

Register or unregister diskio driver for given drive number.

When FATFS library calls one of disk_xxx functions for driver number pdrv, corresponding function in discio_impl for given pdrv will be called.

Parameters

- **pdrv** –drive number
- **discio_impl** –pointer to *ff_diskio_impl_t* structure with diskio functions or NULL to unregister and free previously registered drive

struct **ff_diskio_impl_t**

Structure of pointers to disk IO driver functions.

See FatFs documentation for details about these functions

Public Members

DSTATUS (***init**)(unsigned char pdrv)

disk initialization function

DSTATUS (***status**)(unsigned char pdrv)

disk status check function

DRESULT (***read**)(unsigned char pdrv, unsigned char *buff, uint32_t sector, unsigned count)

sector read function

DRESULT (***write**)(unsigned char pdrv, const unsigned char *buff, uint32_t sector, unsigned count)

sector write function

DRESULT (***ioctl**)(unsigned char pdrv, unsigned char cmd, void *buff)

function to get info about disk and do some misc operations

void **ff_diskio_register_sdmmc**(unsigned char pdrv, *sdmmc_card_t* *card)

Register SD/MMC diskio driver

Parameters

- **pdrv** –drive number
- **card** –pointer to *sdmmc_card_t* structure describing a card; card should be initialized before calling f_mount.

esp_err_t **ff_diskio_register_wl_partition**(unsigned char pdrv, *wl_handle_t* flash_handle)

Register spi flash partition

Parameters

- **pdrv** –drive number
- **flash_handle** –handle of the wear levelling partition.

esp_err_t **ff_diskio_register_raw_partition**(unsigned char pdrv, const *esp_partition_t* *part_handle)

Register spi flash partition

Parameters

- **pdrv** –drive number
- **part_handle** –pointer to raw flash partition.

FatFs Partition Generator

We provide a partition generator for FatFs (wl_fatfsgen.py) which is integrated into the build system and could be easily used in the user project.

The tool is used to create filesystem images on a host and populate it with content of the specified host folder.

The script is based on the partition generator (fatfsgen.py). Apart from generating partition, it can also initialize wear levelling.

The latest version supports both short and long file names, FAT12 and FAT16. The long file names are limited to 255 characters and can contain multiple periods (`.`) characters within the filename and additional characters +, `,`, `;`, =, `[` and `]`.

Build System Integration with FatFs Partition Generator It is possible to invoke FatFs generator directly from the CMake build system by calling `fatfs_create_spiflash_image`:

```
fatfs_create_spiflash_image(<partition> <base_dir> [FLASH_IN_PROJECT])
```

If you prefer generating partition without wear levelling support, you can use `fatfs_create_rawflash_image`:

```
fatfs_create_rawflash_image(<partition> <base_dir> [FLASH_IN_PROJECT])
```

`fatfs_create_spiflash_image` respectively `fatfs_create_rawflash_image` must be called from project' s CMakeLists.txt.

If you decide for any reason to use `fatfs_create_rawflash_image` (without wear levelling support), beware that it supports mounting only in read-only mode in the device.

The arguments of the function are as follows:

1. partition - the name of the partition as defined in the partition table (e.g. storage/fatfsgen/partitions_example.csv).
2. base_dir - the directory that will be encoded to FatFs partition and optionally flashed into the device. Beware that you have to specify the suitable size of the partition in the partition table.
3. flag `FLASH_IN_PROJECT` - optionally, users can have the image automatically flashed together with the app binaries, partition tables, etc. on `idf.py flash -p <PORT>` by specifying `FLASH_IN_PROJECT`.
4. flag `PRESERVE_TIME` - optionally, users can force preserving the timestamps from the source folder to the target image. Without preserving the time, every timestamp will be set to the FATFS default initial time (1st January 1980).

For example:

```
fatfs_create_spiflash_image(my_fatfs_partition my_folder FLASH_IN_PROJECT)
```

If FLASH_IN_PROJECT is not specified, the image will still be generated, but you will have to flash it manually using `esptool.py` or a custom build system target.

For an example, see storage/fatfsgen.

FatFs Partition Analyzer

(fatfsparse.py) is a partition analyzing tool for FatFs.

It is a reverse tool of (fatfsgen.py), i.e. it can generate the folder structure on the host based on the FatFs image.

Usage:

```
./fatfsparse.py [-h] [--long-name-support] [--wear-leveling] fatfs_image.img
```

High-level API Reference

Header File

- components/fatfs/vfs/esp_vfs_fat.h

Functions

esp_err_t **esp_vfs_fat_register**(const char *base_path, const char *fat_drive, size_t max_files, FATFS **out_fs)

Register FATFS with VFS component.

This function registers given FAT drive in VFS, at the specified base path. If only one drive is used, fat_drive argument can be an empty string. Refer to FATFS library documentation on how to specify FAT drive. This function also allocates FATFS structure which should be used for f_mount call.

Note: This function doesn't mount the drive into FATFS, it just connects POSIX and C standard library IO function with FATFS. You need to mount desired drive into FATFS separately.

Parameters

- **base_path** –path prefix where FATFS should be registered
- **fat_drive** –FATFS drive specification; if only one drive is used, can be an empty string
- **max_files** –maximum number of files which can be open at the same time
- **out_fs** –**[out]** pointer to FATFS structure which can be used for FATFS f_mount call is returned via this argument.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_register was already called
- ESP_ERR_NO_MEM if not enough memory or too many VFSes already registered

esp_err_t **esp_vfs_fat_unregister_path**(const char *base_path)

Un-register FATFS from VFS.

Note: FATFS structure returned by esp_vfs_fat_register is destroyed after this call. Make sure to call f_mount function to unmount it before calling esp_vfs_fat_unregister_ctx. Difference between this function and the one above is that this one will release the correct drive, while the one above will release the last registered one

Parameters **base_path** –path prefix where FATFS is registered. This is the same used when esp_vfs_fat_register was called

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if FATFS is not registered in VFS

esp_err_t **esp_vfs_fat_sdmmc_mount**(const char *base_path, const *sdmmc_host_t* *host_config, const void *slot_config, const *esp_vfs_fat_mount_config_t* *mount_config, *sdmmc_card_t* **out_card)

Convenience function to get FAT filesystem on SD card registered in VFS.

This is an all-in-one function which does the following:

- initializes SDMMC driver or SPI driver with configuration in host_config
- initializes SD card with configuration in slot_config
- mounts FAT partition on SD card using FATFS library, with configuration in mount_config
- registers FATFS library with VFS, with prefix given by base_prefix variable

This function is intended to make example code more compact. For real world applications, developers should implement the logic of probing SD card, locating and mounting partition, and registering FATFS in VFS, with proper error checking and handling of exceptional conditions.

Note: Use this API to mount a card through SDSPI is deprecated. Please call `esp_vfs_fat_sdspi_mount()` instead for that case.

Parameters

- **base_path** –path where partition should be registered (e.g. "/sdcard")
- **host_config** –Pointer to structure describing SDMMC host. When using SDMMC peripheral, this structure can be initialized using SDMMC_HOST_DEFAULT() macro. When using SPI peripheral, this structure can be initialized using SDSPI_HOST_DEFAULT() macro.
- **slot_config** –Pointer to structure with slot configuration. For SDMMC peripheral, pass a pointer to *sdmmc_slot_config_t* structure initialized using SDMMC_SLOT_CONFIG_DEFAULT.
- **mount_config** –pointer to structure with extra parameters for mounting FATFS
- **out_card** –**[out]** if not NULL, pointer to the card information structure will be returned via this argument

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount was already called
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from SDMMC or SPI drivers, SDMMC protocol, or FATFS drivers

esp_err_t **esp_vfs_fat_sdspi_mount**(const char *base_path, const *sdmmc_host_t* *host_config_input, const *sdspi_device_config_t* *slot_config, const *esp_vfs_fat_mount_config_t* *mount_config, *sdmmc_card_t* **out_card)

Convenience function to get FAT filesystem on SD card registered in VFS.

This is an all-in-one function which does the following:

- initializes an SPI Master device based on the SPI Master driver with configuration in slot_config, and attach it to an initialized SPI bus.
- initializes SD card with configuration in host_config_input
- mounts FAT partition on SD card using FATFS library, with configuration in mount_config
- registers FATFS library with VFS, with prefix given by base_prefix variable

This function is intended to make example code more compact. For real world applications, developers should implement the logic of probing SD card, locating and mounting partition, and registering FATFS in VFS, with proper error checking and handling of exceptional conditions.

Note: This function try to attach the new SD SPI device to the bus specified in host_config. Make sure the SPI bus specified in `host_config->slot` have been initialized by `spi_bus_initialize()` before.

Parameters

- **base_path** –path where partition should be registered (e.g. "/sdcard")
- **host_config_input** –Pointer to structure describing SDMMC host. This structure can be initialized using SDSPI_HOST_DEFAULT() macro.
- **slot_config** –Pointer to structure with slot configuration. For SPI peripheral, pass a pointer to *sdspi_device_config_t* structure initialized using SDSPI_DEVICE_CONFIG_DEFAULT().
- **mount_config** –pointer to structure with extra parameters for mounting FATFS
- **out_card** –**[out]** If not NULL, pointer to the card information structure will be returned via this argument. It is suggested to hold this handle and use it to unmount the card later if needed. Otherwise it's not suggested to use more than one card at the same time and unmount one of them in your application.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount was already called
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from SDMMC or SPI drivers, SDMMC protocol, or FATFS drivers

esp_err_t **esp_vfs_fat_sdmmc_unmount**(void)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_sdmmc_mount.

Deprecated:

Use `esp_vfs_fat_sdcard_unmount()` instead.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount hasn't been called

esp_err_t **esp_vfs_fat_sdcard_unmount**(const char *base_path, *sdmmc_card_t* *card)

Unmount an SD card from the FAT filesystem and release resources acquired using `esp_vfs_fat_sdmmc_mount()` or `esp_vfs_fat_sdspi_mount()`

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the card argument is unregistered
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount hasn't been called

esp_err_t **esp_vfs_fat_spiflash_mount_rw_wl**(const char *base_path, const char *partition_label, const *esp_vfs_fat_mount_config_t* *mount_config, *wl_handle_t* *wl_handle)

Convenience function to initialize FAT filesystem in SPI flash and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined partition_label. Partition label should be configured in the partition table.
- initializes flash wear levelling library on top of the given partition
- mounts FAT partition using FATFS library on top of flash wear levelling library
- registers FATFS library with VFS, with prefix given by base_prefix variable

This function is intended to make example code more compact.

Parameters

- **base_path** –path where FATFS partition should be mounted (e.g. "/spiflash")
- **partition_label** –label of the partition which should be used
- **mount_config** –pointer to structure with extra parameters for mounting FATFS
- **wl_handle** –**[out]** wear levelling driver handle

Returns

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition table does not contain FATFS partition with given label
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount_rw_wl was already called
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from wear levelling library, SPI flash driver, or FATFS drivers

esp_err_t **esp_vfs_fat_spiflash_unmount_rw_wl**(const char *base_path, *wl_handle_t* wl_handle)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_spiflash_mount_rw_wl.

Parameters

- **base_path** –path where partition should be registered (e.g. "/spiflash")
- **wl_handle** –wear levelling driver handle returned by esp_vfs_fat_spiflash_mount_rw_wl

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount_rw_wl hasn't been called

esp_err_t **esp_vfs_fat_spiflash_mount_ro**(const char *base_path, const char *partition_label, const *esp_vfs_fat_mount_config_t* *mount_config)

Convenience function to initialize read-only FAT filesystem and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined partition_label. Partition label should be configured in the partition table.
- mounts FAT partition using FATFS library
- registers FATFS library with VFS, with prefix given by base_prefix variable

Note: Wear levelling is not used when FAT is mounted in read-only mode using this function.

Parameters
- **base_path** –path where FATFS partition should be mounted (e.g. "/spiflash")
- **partition_label** –label of the partition which should be used
- **mount_config** –pointer to structure with extra parameters for mounting FATFS

Returns
- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition table does not contain FATFS partition with given label
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount_ro was already called for the same partition
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from SPI flash driver, or FATFS drivers

esp_err_t **esp_vfs_fat_spiflash_unmount_ro**(const char *base_path, const char *partition_label)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_spiflash_mount_ro.

Parameters
- **base_path** –path where partition should be registered (e.g. "/spiflash")
- **partition_label** –label of partition to be unmounted

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount_rw_wl hasn't been called

esp_err_t **esp_vfs_fat_info**(const char *base_path, uint64_t *out_total_bytes, uint64_t *out_free_bytes)

Get information for FATFS partition.

Parameters
- **base_path** –Base path of the partition examined (e.g. "/spiflash")
- **out_total_bytes** –**[out]** Size of the file system
- **out_free_bytes** –**[out]** Free bytes available in the file system

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if partition not found
- ESP_FAIL if another FRESULT error (saved in errno)

Structures

struct **esp_vfs_fat_mount_config_t**

Configuration arguments for esp_vfs_fat_sdmmc_mount and esp_vfs_fat_spiflash_mount_rw_wl functions.

Public Members

bool **format_if_mount_failed**

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

int **max_files**

Max number of open files.

size_t **allocation_unit_size**

If format_if_mount_failed is set, and mount fails, format the card with given allocation unit size. Must be a power of 2, between sector size and 128 * sector size. For SD cards, sector size is always 512 bytes. For wear_levelling, sector size is determined by CONFIG_WL_SECTOR_SIZE option.

Using larger allocation unit size will result in higher read/write performance and higher overhead when storing small files.

Setting this field to 0 will result in allocation unit set to the sector size.

bool **disk_status_check_enable**

Enables real ff_disk_status function implementation for SD cards (ff_sdmmc_status). Possibly slows down IO performance.

Try to enable if you need to handle situations when SD cards are not unmounted properly before physical removal or you are experiencing issues with SD cards.

Doesn't do anything for other memory storage media.

Type Definitions

typedef *esp_vfs_fat_mount_config_t* **esp_vfs_fat_sdmmc_mount_config_t**

2.9.2 Manufacturing Utility

Introduction

This utility is designed to create instances of factory NVS partition images on a per-device basis for mass manufacturing purposes. The NVS partition images are created from CSV files containing user-provided configurations and values.

Please note that this utility only creates manufacturing binary images which then need to be flashed onto your devices using:

- esptool.py
- Flash Download tool (available on Windows only).Just download it, unzip, and follow the instructions inside the *doc* folder.
- Direct flash programming using custom production tools.

Prerequisites

This utility is dependent on esp-idf's NVS partition utility.

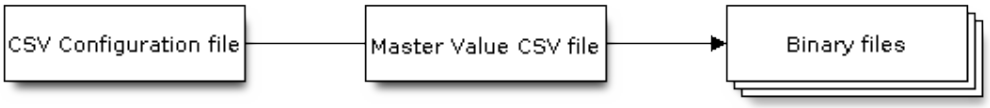
- **Operating System requirements:**
  - Linux / MacOS / Windows (standard distributions)
- **The following packages are needed to use this utility:**
  - Python

Note:

Before using this utility, please make sure that:

- The path to Python is added to the PATH environment variable.
- You have installed the packages from *requirement.txt*, the file in the root of the esp-idf directory.

Workflow



CSV Configuration File

This file contains the configuration of the device to be flashed.

The data in the configuration file has the following format (the *REPEAT* tag is optional):

```
name1,namespace,    <-- First entry should be of type "namespace"
key1,type1,encoding1
key2,type2,encoding2,REPEAT
name2,namespace,
key3,type3,encoding3
key4,type4,encoding4
```

Note: The first line in this file should always be the `namespace` entry.

Each line should have three parameters: `key,type,encoding`, separated by a comma. If the `REPEAT` tag is present, the value corresponding to this key in the master value CSV file will be the same for all devices.

Please refer to README of the NVS Partition Generator utility for detailed description of each parameter.

Below is a sample example of such a configuration file:

```
app,namespace,
firmware_key,data,hex2bin
serial_no,data,string,REPEAT
device_no,data,i32
```

Note:

Make sure there are no spaces:

- before and after ‘,’
- at the end of each line in a CSV file

Master Value CSV File

This file contains details of the devices to be flashed. Each line in this file corresponds to a device instance.

The data in the master value CSV file has the following format:

```
key1,key2,key3,.....
value1,value2,value3,....
```

Note: The first line in the file should always contain the `key` names. All the keys from the configuration file should be present here in the **same order**. This file can have additional columns (keys). The additional keys will be treated as metadata and would not be part of the final binary files.

Each line should contain the `value` of the corresponding keys, separated by a comma. If the key has the `REPEAT` tag, its corresponding value **must** be entered in the second line only. Keep the entry empty for this value in the following lines.

The description of this parameter is as follows:

`value` Data value

Data value is the value of data corresponding to the key.

Below is a sample example of a master value CSV file:

```
id,firmware_key,serial_no,device_no
1,1a2b3c4d5e6faabb,A1,101
2,1a2b3c4d5e6fccdd,,102
3,1a2b3c4d5e6feeff,,103
```

Note: *If the 'REPEAT' tag is present, a new master value CSV file will be created in the same folder as the input Master CSV File with the values inserted at each line for the key with the 'REPEAT' tag.*

This utility creates intermediate CSV files which are used as input for the NVS partition utility to generate the binary files.

The format of this intermediate CSV file is as follows:

```
key,type,encoding,value
key,namespace, ,
key1,type1,encoding1,value1
key2,type2,encoding2,value2
```

An instance of an intermediate CSV file will be created for each device on an individual basis.

Running the utility

Usage:

```
python mfg_gen.py [-h] {generate,generate-key} ...
```

Optional Arguments:

No.	Parameter	Description
1	-h, –help	show this help message and exit

Commands:

Run mfg_gen.py {command} -h for additional help

No.	Parameter	Description
1	generate	Generate NVS partition
2	generate-key	Generate keys for encryption

To generate factory images for each device (Default):

Usage:

```
python mfg_gen.py generate [-h] [--fileid FILEID] [--version {1,2}] [--keygen]
                                [--keyfile KEYFILE] [--inputkey INPUTKEY]
                                [--outdir OUTDIR]
                                conf values prefix size
```

Positional Arguments:

Parameter	Description
conf	Path to configuration csv file to parse
values	Path to values csv file to parse
prefix | Unique name for each output filename prefix	
size | Size of NVS partition in bytes (must be multiple of 4096)	

Optional Arguments:

Parameter	Description
-h, –help	show this help message and exit
–fileid FILEID	Unique file identifier(any key in values file) for each filename suffix (Default: numeric value(1,2,3⋯)
–version {1,2}	Set multipage blob version. Version 1 - Multipage blob support disabled. Version 2 - Multipage blob support enabled. Default: Version 2
–keygen	Generates key for encrypting NVS partition
–inputkey INPUTKEY	File having key for encrypting NVS partition
–outdir OUTDIR	Output directory to store files created (Default: current directory)

You can run the utility to generate factory images for each device using the command below. A sample CSV file is provided with the utility:

```
python mfg_gen.py generate samples/sample_config.csv samples/sample_values_
↪singlepage_blob.csv Sample 0x3000
```

The master value CSV file should have the path in the `file` type relative to the directory from which you are running the utility.

To generate encrypted factory images for each device:

You can run the utility to encrypt factory images for each device using the command below. A sample CSV file is provided with the utility:

- Encrypt by allowing the utility to generate encryption keys:

```
python mfg_gen.py generate samples/sample_config.csv samples/sample_values_
↪singlepage_blob.csv Sample 0x3000 --keygen
```

Note: Encryption key of the following format `<outdir>/keys/keys-<prefix>-<fileid>.bin` is created. This newly created file having encryption keys in `keys/` directory is compatible with NVS key-partition structure. Refer to *NVS Key Partition* for more details.

- Encrypt by providing the encryption keys as input binary file:

```
python mfg_gen.py generate samples/sample_config.csv samples/sample_values_
↪singlepage_blob.csv Sample 0x3000 --inputkey keys/sample_keys.bin
```

To generate only encryption keys:

Usage:: python mfg_gen.py generate-key [-h] [–keyfile KEYFILE] [–outdir OUTDIR]
Optional Arguments:

Parameter	Description
-h, –help	show this help message and exit
–keyfile KEYFILE	Path to output encryption keys file
–outdir OUTDIR	Output directory to store files created. (Default: current directory)

You can run the utility to generate only encryption keys using the command below:

```
python mfg_gen.py generate-key
```

Note: Encryption key of the following format `<outdir>/keys/keys-<timestamp>.bin` is created. Timestamp format is: `%m-%d_%H-%M`. To provide custom target filename use the –keyfile argument.

Generated encryption key binary file can further be used to encrypt factory images created on the per device basis.

The default numeric value: 1,2,3⋯ of the `fileid` argument corresponds to each line bearing device instance values in the master value CSV file.

While running the manufacturing utility, the following folders will be created in the specified `outdir` directory:

- `bin/` for storing the generated binary files
- `csv/` for storing the generated intermediate CSV files
- `keys/` for storing encryption keys (when generating encrypted factory images)

2.9.3 Non-volatile Storage Library

Introduction

Non-volatile storage (NVS) library is designed to store key-value pairs in flash. This section introduces some concepts used by NVS.

Underlying Storage Currently, NVS uses a portion of main flash memory through the *esp_partition* API. The library uses all the partitions with `data` type and `nvs` subtype. The application can choose to use the partition with the label `nvs` through the *`nvs_open()`* API function or any other partition by specifying its name using the *`nvs_open_from_partition()`* API function.

Future versions of this library may have other storage backends to keep data in another flash chip (SPI or I2C), RTC, FRAM, etc.

Note: if an NVS partition is truncated (for example, when the partition table layout is changed), its contents should be erased. ESP-IDF build system provides a `idf.py erase-flash` target to erase all contents of the flash chip.

Note: NVS works best for storing many small values, rather than a few large values of the type ‘string’ and ‘blob’. If you need to store large blobs or strings, consider using the facilities provided by the FAT filesystem on top of the wear levelling library.

Keys and Values NVS operates on key-value pairs. Keys are ASCII strings; the maximum key length is currently 15 characters. Values can have one of the following types:

- integer types: `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t`, `int64_t`
- zero-terminated string
- variable length binary data (blob)

Note: String values are currently limited to 4000 bytes. This includes the null terminator. Blob values are limited to 508,000 bytes or 97.6% of the partition size - 4000 bytes, whichever is lower.

Additional types, such as `float` and `double` might be added later.

Keys are required to be unique. Assigning a new value to an existing key works as follows:

- If the new value is of the same type as the old one, value is updated.
- If the new value has a different data type, an error is returned.

Data type check is also performed when reading a value. An error is returned if the data type of the read operation does not match the data type of the value.

Namespaces To mitigate potential conflicts in key names between different components, NVS assigns each key-value pair to one of namespaces. Namespace names follow the same rules as key names, i.e., the maximum length is 15 characters. Furthermore, there can be no more than 254 different namespaces in one NVS partition. Namespace name is specified in the *`nvs_open()`* or *`nvs_open_from_partition`* call. This call returns an opaque handle, which is used in subsequent calls to the `nvs_get_*`, `nvs_set_*`, and *`nvs_commit()`* functions. This way, a handle is associated with a namespace, and key names will not collide with same names in other namespaces. Please note that the namespaces with the same name in different NVS partitions are considered as separate namespaces.

NVS Iterators Iterators allow to list key-value pairs stored in NVS, based on specified partition name, namespace, and data type.

There are the following functions available:

- *`nvs_entry_find()`* creates an opaque handle, which is used in subsequent calls to the *`nvs_entry_next()`* and *`nvs_entry_info()`* functions.
- *`nvs_entry_next()`* advances an iterator to the next key-value pair.
- *`nvs_entry_info()`* returns information about each key-value pair

In general, all iterators obtained via *`nvs_entry_find()`* have to be released using *`nvs_release_iterator()`*, which also tolerates `NULL` iterators. *`nvs_entry_find()`* and *`nvs_entry_next()`* will set the given iterator to `NULL` or a valid iterator in all cases except a parameter error occured (i.e., return `ESP_ERR_NVS_NOT_FOUND`). In case of a parameter error, the given iterator will not be modified. Hence, it is best practice to initialize the iterator to `NULL` before calling *`nvs_entry_find()`* to avoid complicated error checking before releasing the iterator.

Security, Tampering, and Robustness NVS is not directly compatible with the ESP32 flash encryption system. However, data can still be stored in encrypted form if NVS encryption is used together with ESP32 flash encryption. Please refer to *NVS Encryption* for more details.

If NVS encryption is not used, it is possible for anyone with physical access to the flash chip to alter, erase, or add key-value pairs. With NVS encryption enabled, it is not possible to alter or add a key-value pair and get recognized as a valid pair without knowing corresponding NVS encryption keys. However, there is no tamper-resistance against the erase operation.

The library does try to recover from conditions when flash memory is in an inconsistent state. In particular, one should be able to power off the device at any point and time and then power it back on. This should not result in loss of data, except for the new key-value pair if it was being written at the moment of powering off. The library should also be able to initialize properly with any random data present in flash memory.

NVS Encryption

Data stored in NVS partitions can be encrypted using AES-XTS in the manner similar to the one mentioned in disk encryption standard IEEE P1619. For the purpose of encryption, each entry is treated as one *sector* and relative address of the entry (w.r.t. partition-start) is fed to the encryption algorithm as *sector-number*. The NVS Encryption can be enabled by enabling *CONFIG_NVS_ENCRYPTION*. The keys required for NVS encryption are stored in yet another partition, which is protected using *Flash Encryption*. Therefore, enabling *Flash Encryption* is a prerequisite for NVS encryption.

The NVS Encryption is enabled by default when *Flash Encryption* is enabled. This is done because Wi-Fi driver stores credentials (like SSID and passphrase) in the default NVS partition. It is important to encrypt them as default choice if platform level encryption is already enabled.

For using NVS encryption, the partition table must contain the *NVS Key Partition*. Two partition tables containing the *NVS Key Partition* are provided for NVS encryption under the partition table option (`menuconfig > Partition Table`). They can be selected with the project configuration menu (`idf.py menuconfig`). Please refer to the example security/flash_encryption for how to configure and use NVS encryption feature.

NVS Key Partition An application requiring NVS encryption support needs to be compiled with a key-partition of the type *data* and subtype *key*. This partition should be marked as *encrypted* and its size should be the minimum partition size (4KB). Refer to *Partition Tables* for more details. Two additional partition tables which contain the *NVS Key Partition* are provided under the partition table option (`menuconfig > Partition Table`). They can be directly used for *NVS Encryption*. The structure of these partitions is depicted below.

```
+-----------+--------------+-------------+----+
|               XTS encryption key (32)        |
+----------------------------------------------+
|               XTS tweak key (32)             |
+----------------------------------------------+
|                   CRC32 (4)                  |
+----------------------------------------------+
```

The XTS encryption keys in the *NVS Key Partition* can be generated in one of the following two ways.

1. Generate the keys on the ESP chip:

   When NVS encryption is enabled the `nvs_flash_init()` API function can be used to initialize the encrypted default NVS partition. The API function internally generates the XTS encryption keys on the ESP chip. The API function finds the first *NVS Key Partition*. Then the API function automatically generates and stores the NVS keys in that partition by making use of the `nvs_flash_generate_keys()` API function provided by nvs_flash/include/nvs_flash.h. New keys are generated and stored only when the respective key partition is empty. The same key partition can then be used to read the security configurations for initializing a custom encrypted NVS partition with help of `nvs_flash_secure_init_partition()`.
   The API functions `nvs_flash_secure_init()` and `nvs_flash_secure_init_partition()` do not generate the keys internally. When these API functions are used for initializing encrypted NVS partitions, the keys can be generated after startup using the `nvs_flash_generate_keys()` API function provided by `nvs_flash.h`. The API function will then write those keys onto the key-partition in encrypted form.

   Note: Please note that *nvs_keys* partition must be completely erased before starting the application in this approach. Otherwise the application may generate `ESP_ERR_NVS_CORRUPT_KEY_PART` error code assuming that *nvs_keys* partition was not empty and contains malformatted data. You can use the following command for this:

   ```
   parttool.py --port PORT --partition-table-file=PARTITION_TABLE_FILE --
   ↪partition-table-offset PARTITION_TABLE_OFFSET erase_partition --
   ↪partition-type=data --partition-subtype=nvs_keys
   ```

   ---

2. Use pre-generated key partition:

This option will be required by the user when keys in the *NVS Key Partition* are not generated by the application. The *NVS Key Partition* containing the XTS encryption keys can be generated with the help of *NVS Partition Generator Utility*. Then the user can store the pre generated key partition on the flash with help of the following two commands:
i) Build and flash the partition table

```
idf.py partition-table partition-table-flash
```

ii) Store the keys in the *NVS Key Partition* (on the flash) with the help of parttool.py (see Partition Tool section in *partition-tables* for more details)

```
parttool.py --port PORT --partition-table-offset PARTITION_TABLE_
↪OFFSET write_partition --partition-name="name of nvs_key partition" -
↪-input NVS_KEY_PARTITION_FILE
```

Note: If the device is encrypted in flash encryption development mode and you want to renew the NVS key partition, you need to tell parttool.py to encrypt the NVS key partition and you also need to give it a pointer to the unencrypted partition table in your build directory (build/partition_table) since the partition table on the device is encrypted, too. You can use the following command:

```
parttool.py --esptool-write-args encrypt --port PORT --partition-table-
↪file=PARTITION_TABLE_FILE --partition-table-offset PARTITION_TABLE_
↪OFFSET write_partition --partition-name="name of nvs_key partition" -
↪-input NVS_KEY_PARTITION_FILE
```

Since the key partition is marked as *encrypted* and *Flash Encryption* is enabled, the bootloader will encrypt this partition using flash encryption key on the first boot.

It is possible for an application to use different keys for different NVS partitions and thereby have multiple key-partitions. However, it is a responsibility of the application to provide correct key-partition/keys for the purpose of encryption/decryption.

Encrypted Read/Write The same NVS API functions `nvs_get_*` or `nvs_set_*` can be used for reading of, and writing to an encrypted nvs partition as well.

Encrypt the default NVS partition: To enable encryption for the default NVS partition no additional steps are necessary. When *CONFIG_NVS_ENCRYPTION* is enabled, the *`nvs_flash_init()`* API function internally performs some additional steps using the first *NVS Key Partition* found to enable encryption for the default NVS partition (refer to the API documentation for more details). Alternatively, *`nvs_flash_secure_init()`* API function can also be used to enable encryption for the default NVS partition.

Encrypt a custom NVS partition: To enable encryption for a custom NVS partition, *`nvs_flash_secure_init_partition()`* API function is used instead of *`nvs_flash_init_partition()`*.

When *`nvs_flash_secure_init()`* and *`nvs_flash_secure_init_partition()`* API functions are used, the applications are expected to follow the steps below in order to perform NVS read/write operations with encryption enabled.

1. Find key partition and NVS data partition using `esp_partition_find*` API functions.
2. Populate the *`nvs_sec_cfg_t`* struct using the *`nvs_flash_read_security_cfg()`* or *`nvs_flash_generate_keys()`* API functions.
3. Initialise NVS flash partition using the *`nvs_flash_secure_init()`* or *`nvs_flash_secure_init_partition()`* API functions.
4. Open a namespace using the *`nvs_open()`* or *`nvs_open_from_partition()`* API functions.
5. Perform NVS read/write operations using `nvs_get_*` or `nvs_set_*`.
6. Deinitialise an NVS partition using *`nvs_flash_deinit()`*.

NVS Partition Generator Utility

This utility helps generate NVS partition binary files which can be flashed separately on a dedicated partition via a flashing utility. Key-value pairs to be flashed onto the partition can be provided via a CSV file. For more details, please refer to *NVS Partition Generator Utility*.

Application Example

You can find code examples in the storage directory of ESP-IDF examples:

storage/nvs_rw_value

Demonstrates how to read a single integer value from, and write it to NVS.

The value checked in this example holds the number of the ESP32 module restarts. The value ' s function as a counter is only possible due to its storing in NVS.

The example also shows how to check if a read / write operation was successful, or if a certain value has not been initialized in NVS. The diagnostic procedure is provided in plain text to help you track the program flow and capture any issues on the way.

storage/nvs_rw_blob

Demonstrates how to read a single integer value and a blob (binary large object), and write them to NVS to preserve this value between ESP32 module restarts.

- value - tracks the number of the ESP32 module soft and hard restarts.
- blob - contains a table with module run times. The table is read from NVS to dynamically allocated RAM. A new run time is added to the table on each manually triggered soft restart, and then the added run time is written to NVS. Triggering is done by pulling down GPIO0.

The example also shows how to implement the diagnostic procedure to check if the read / write operation was successful.

storage/nvs_rw_value_cxx

This example does exactly the same as storage/nvs_rw_value, except that it uses the C++ NVS handle class.

Internals

Log of Key-Value Pairs NVS stores key-value pairs sequentially, with new key-value pairs being added at the end. When a value of any given key has to be updated, a new key-value pair is added at the end of the log and the old key-value pair is marked as erased.

Pages and Entries NVS library uses two main entities in its operation: pages and entries. Page is a logical structure which stores a portion of the overall log. Logical page corresponds to one physical sector of flash memory. Pages which are in use have a *sequence number* associated with them. Sequence numbers impose an ordering on pages. Higher sequence numbers correspond to pages which were created later. Each page can be in one of the following states:

Empty/uninitialized Flash storage for the page is empty (all bytes are `0xff`). Page is not used to store any data at this point and does not have a sequence number.

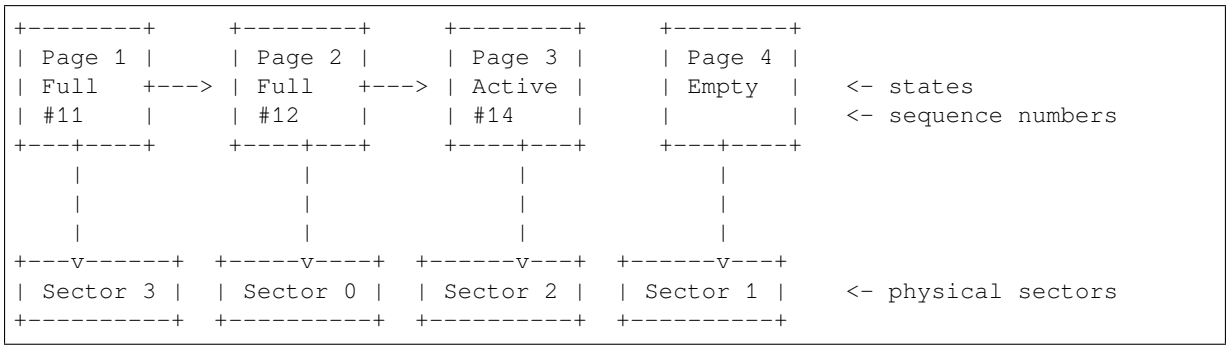
Active Flash storage is initialized, page header has been written to flash, page has a valid sequence number. Page has some empty entries and data can be written there. No more than one page can be in this state at any given moment.

Full Flash storage is in a consistent state and is filled with key-value pairs. Writing new key-value pairs into this page is not possible. It is still possible to mark some key-value pairs as erased.

Erasing Non-erased key-value pairs are being moved into another page so that the current page can be erased. This is a transient state, i.e., page should never stay in this state at the time when any API call returns. In case of a sudden power off, the move-and-erase process will be completed upon the next power-on.

Corrupted Page header contains invalid data, and further parsing of page data was canceled. Any items previously written into this page will not be accessible. The corresponding flash sector will not be erased immediately and will be kept along with sectors in *uninitialized* state for later use. This may be useful for debugging.

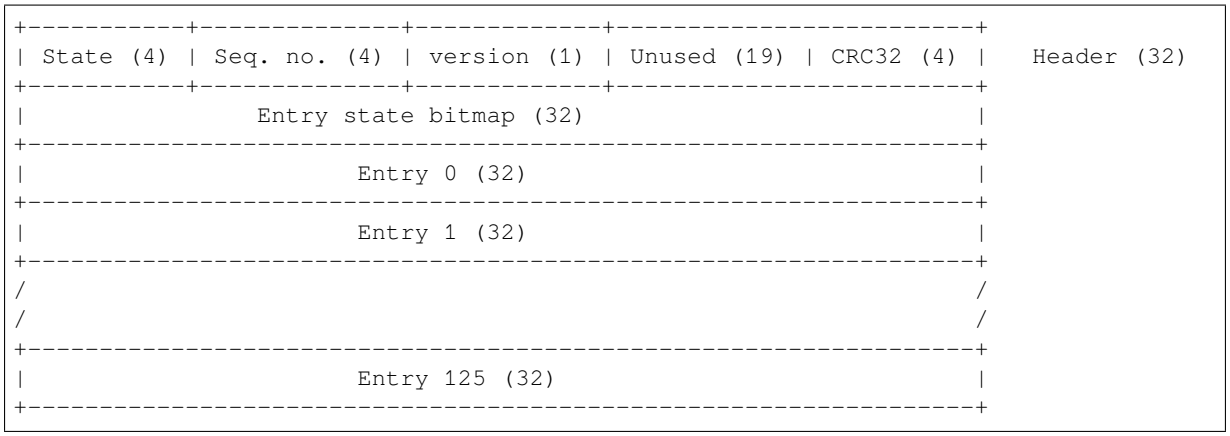
Mapping from flash sectors to logical pages does not have any particular order. The library will inspect sequence numbers of pages found in each flash sector and organize pages in a list based on these numbers.



Structure of a Page For now, we assume that flash sector size is 4096 bytes and that ESP32 flash encryption hardware operates on 32-byte blocks. It is possible to introduce some settings configurable at compile-time (e.g., via menuconfig) to accommodate flash chips with different sector sizes (although it is not clear if other components in the system, e.g., SPI flash driver and SPI flash cache can support these other sizes).

Page consists of three parts: header, entry state bitmap, and entries themselves. To be compatible with ESP32 flash encryption, the entry size is 32 bytes. For integer types, an entry holds one key-value pair. For strings and blobs, an entry holds part of key-value pair (more on that in the entry structure description).

The following diagram illustrates the page structure. Numbers in parentheses indicate the size of each part in bytes.



Page header and entry state bitmap are always written to flash unencrypted. Entries are encrypted if flash encryption feature of ESP32 is used.

Page state values are defined in such a way that changing state is possible by writing 0 into some of the bits. Therefore it is not necessary to erase the page to change its state unless that is a change to the *erased* state.

The version field in the header reflects the NVS format version used. For backward compatibility reasons, it is decremented for every version upgrade starting at 0xff (i.e., 0xff for version-1, 0xfe for version-2 and so on).

CRC32 value in the header is calculated over the part which does not include a state value (bytes 4 to 28). The unused part is currently filled with `0xff` bytes.

The following sections describe the structure of entry state bitmap and entry itself.

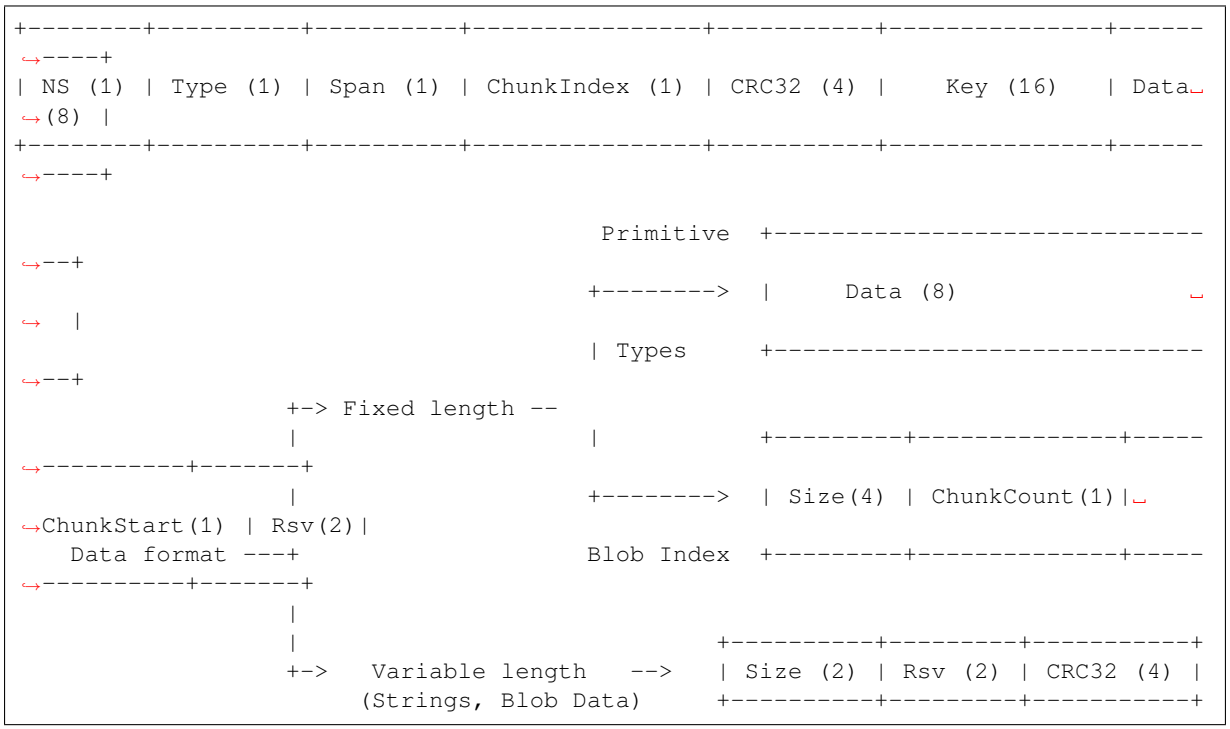
Entry and Entry State Bitmap Each entry can be in one of the following three states represented with two bits in the entry state bitmap. The final four bits in the bitmap (256 - 2 * 126) are not used.

Empty (2' b11) Nothing is written into the specific entry yet. It is in an uninitialized state (all bytes are `0xff`).

Written (2' b10) A key-value pair (or part of key-value pair which spans multiple entries) has been written into the entry.

Erased (2' b00) A key-value pair in this entry has been discarded. Contents of this entry will not be parsed anymore.

Structure of Entry For values of primitive types (currently integers from 1 to 8 bytes long), entry holds one key-value pair. For string and blob types, entry holds part of the whole key-value pair. For strings, in case when a key-value pair spans multiple entries, all entries are stored in the same page. Blobs are allowed to span over multiple pages by dividing them into smaller chunks. For tracking these chunks, an additional fixed length metadata entry is stored called "blob index". Earlier formats of blobs are still supported (can be read and modified). However, once the blobs are modified, they are stored using the new format.



Individual fields in entry structure have the following meanings:

NS Namespace index for this entry. For more information on this value, see the section on namespaces implementation.

Type One byte indicating the value data type. See the `ItemType` enumeration in nvs_flash/include/nvs_handle.hpp for possible values.

Span Number of entries used by this key-value pair. For integer types, this is equal to 1. For strings and blobs, this depends on value length.

ChunkIndex Used to store the index of a blob-data chunk for blob types. For other types, this should be `0xff`.

CRC32 Checksum calculated over all the bytes in this entry, except for the CRC32 field itself.

Key Zero-terminated ASCII string containing a key name. Maximum string length is 15 bytes, excluding a zero terminator.

Data For integer types, this field contains the value itself. If the value itself is shorter than 8 bytes, it is padded to the right, with unused bytes filled with `0xff`.

For "blob index" entry, these 8 bytes hold the following information about data-chunks:

- **Size** (Only for blob index.) Size, in bytes, of complete blob data.
- **ChunkCount** (Only for blob index.) Total number of blob-data chunks into which the blob was divided during storage.
- **ChunkStart** (Only for blob index.) ChunkIndex of the first blob-data chunk of this blob. Subsequent chunks have chunkIndex incrementally allocated (step of 1).

For string and blob data chunks, these 8 bytes hold additional data about the value, which are described below:

- **Size** (Only for strings and blobs.) Size, in bytes, of actual data. For strings, this includes zero terminators.
- **CRC32** (Only for strings and blobs.) Checksum calculated over all bytes of data.

Variable length values (strings and blobs) are written into subsequent entries, 32 bytes per entry. The *Span* field of the first entry indicates how many entries are used.

Namespaces As mentioned above, each key-value pair belongs to one of the namespaces. Namespace identifiers (strings) are stored as keys of key-value pairs in namespace with index 0. Values corresponding to these keys are indexes of these namespaces.

```
+-----------------------------------------+
| NS=0 Type=uint8_t Key="wifi" Value=1    |   Entry describing namespace "wifi"
+-----------------------------------------+
| NS=1 Type=uint32_t Key="channel" Value=6  |   Key "channel" in namespace "wifi"
+-----------------------------------------+
| NS=0 Type=uint8_t Key="pwm" Value=2     |   Entry describing namespace "pwm"
+-----------------------------------------+
| NS=2 Type=uint16_t Key="channel" Value=20 |   Key "channel" in namespace "pwm"
+-----------------------------------------+
```

Item Hash List To reduce the number of reads from flash memory, each member of the Page class maintains a list of pairs: item index; item hash. This list makes searches much quicker. Instead of iterating over all entries, reading them from flash one at a time, *Page::findItem* first performs a search for the item hash in the hash list. This gives the item index within the page if such an item exists. Due to a hash collision, it is possible that a different item will be found. This is handled by falling back to iteration over items in flash.

Each node in the hash list contains a 24-bit hash and 8-bit item index. Hash is calculated based on item namespace, key name, and ChunkIndex. CRC32 is used for calculation; the result is truncated to 24 bits. To reduce the overhead for storing 32-bit entries in a linked list, the list is implemented as a double-linked list of arrays. Each array holds 29 entries, for the total size of 128 bytes, together with linked list pointers and a 32-bit count field. The minimum amount of extra RAM usage per page is therefore 128 bytes; maximum is 640 bytes.

API Reference

Header File

- components/nvs_flash/include/nvs_flash.h

Functions

esp_err_t **nvs_flash_init** (void)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled "nvs" in the partition table.

When "NVS_ENCRYPTION" is enabled in the menuconfig, this API enables the NVS encryption for the default NVS partition as follows

a. Read security configurations from the first NVS key partition listed in the partition table. (NVS key partition is any "data" type partition which has the subtype value set to "nvs_keys")
b. If the NVS key partiton obtained in the previous step is empty, generate and store new keys in that NVS key partiton.
c. Internally call "nvs_flash_secure_init()" with the security configurations obtained/generated in the previous steps.

Post initialization NVS read/write APIs remain the same irrespective of NVS encryption.

Returns

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if no partition with label "nvs" is found in the partition table

- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver
- error codes from nvs_flash_read_security_cfg API (when "NVS_ENCRYPTION" is enabled).
- error codes from nvs_flash_generate_keys API (when "NVS_ENCRYPTION" is enabled).
- error codes from nvs_flash_secure_init_partition API (when "NVS_ENCRYPTION" is enabled) .

esp_err_t **nvs_flash_init_partition**(const char *partition_label)

Initialize NVS flash storage for the specified partition.

Parameters **partition_label** –**[in]** Label of the partition. Must be no longer than 16 characters.

Returns

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if specified partition is not found in the partition table
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_init_partition_ptr**(const *esp_partition_t* *partition)

Initialize NVS flash storage for the partition specified by partition pointer.

Parameters **partition** –**[in]** pointer to a partition obtained by the ESP partition API.

Returns

- ESP_OK if storage was successfully initialized
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_INVALID_ARG in case partition is NULL
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_deinit**(void)

Deinitialize NVS storage for the default NVS partition.

Default NVS partition is the partition with "nvs" label in the partition table.

Returns

- ESP_OK on success (storage was deinitialized)
- ESP_ERR_NVS_NOT_INITIALIZED if the storage was not initialized prior to this call

esp_err_t **nvs_flash_deinit_partition**(const char *partition_label)

Deinitialize NVS storage for the given NVS partition.

Parameters **partition_label** –**[in]** Label of the partition

Returns

- ESP_OK on success
- ESP_ERR_NVS_NOT_INITIALIZED if the storage for given partition was not initialized prior to this call

esp_err_t **nvs_flash_erase**(void)

Erase the default NVS partition.

Erases all contents of the default NVS partition (one with label "nvs").

Note: If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Returns

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition labeled “nvs” in the partition table
- different error in case de-initialization fails (shouldn’t happen)

esp_err_t **nvs_flash_erase_partition** (const char *part_name)

Erase specified NVS partition.

Erase all content of a specified NVS partition

Note: If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Parameters **part_name** –**[in]** Name (label) of the partition which should be erased

Returns

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition with the specified name in the partition table
- different error in case de-initialization fails (shouldn’t happen)

esp_err_t **nvs_flash_erase_partition_ptr** (const *esp_partition_t* *partition)

Erase custom partition.

Erase all content of specified custom partition.

Note: If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Parameters **partition** –**[in]** pointer to a partition obtained by the ESP partition API.

Returns

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no partition with the specified parameters in the partition table
- ESP_ERR_INVALID_ARG in case partition is NULL
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_secure_init** (*nvs_sec_cfg_t* *cfg)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled “nvs” in the partition table.

Parameters **cfg** –**[in]** Security configuration (keys) to be used for NVS encryption/decryption. If cfg is NULL, no encryption is used.

Returns

- ESP_OK if storage has been initialized successfully.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if no partition with label “nvs” is found in the partition table
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_secure_init_partition** (const char *partition_label, *nvs_sec_cfg_t* *cfg)

Initialize NVS flash storage for the specified partition.

Parameters

- **partition_label** –**[in]** Label of the partition. Note that internally, a reference to passed value is kept and it should be accessible for future operations
- **cfg** –**[in]** Security configuration (keys) to be used for NVS encryption/decryption. If cfg is null, no encryption/decryption is used.

Returns

- ESP_OK if storage has been initialized successfully.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if specified partition is not found in the partition table
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_generate_keys**(const *esp_partition_t* *partition, *nvs_sec_cfg_t* *cfg)

Generate and store NVS keys in the provided esp partition.

Parameters

- **partition** –**[in]** Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **cfg** –**[out]** Pointer to nvs security configuration structure. Pointer must be non-NULL. Generated keys will be populated in this structure.

Returns -ESP_OK, if cfg was read successfully; -ESP_INVALID_ARG, if partition or cfg; -or error codes from esp_partition_write/erase APIs.

esp_err_t **nvs_flash_read_security_cfg**(const *esp_partition_t* *partition, *nvs_sec_cfg_t* *cfg)

Read NVS security configuration from a partition.

Note: Provided partition is assumed to be marked ‘encrypted’.

Parameters

- **partition** –**[in]** Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **cfg** –**[out]** Pointer to nvs security configuration structure. Pointer must be non-NULL.

Returns -ESP_OK, if cfg was read successfully; -ESP_INVALID_ARG, if partition or cfg; -ESP_ERR_NVS_KEYS_NOT_INITIALIZED, if the partition is not yet written with keys. -ESP_ERR_NVS_CORRUPT_KEY_PART, if the partition containing keys is found to be corrupt -or error codes from esp_partition_read API.

Structures

struct **nvs_sec_cfg_t**

Key for encryption and decryption.

Public Members

uint8_t **eky**[NVS_KEY_SIZE]

XTS encryption and decryption key

uint8_t **tky**[NVS_KEY_SIZE]

XTS tweak key

Macros

NVS_KEY_SIZE

Header File

- components/nvs_flash/include/nvs.h

Functions

esp_err_t **nvs_set_i8** (*nvs_handle_t* handle, const char *key, int8_t value)

set int8_t value for given key

Set value for the key, given its name. Note that the actual storage will not be updated until `nvs_commit` is called.

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **value** –**[in]** The value to set.

Returns

- ESP_OK if value was set successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if storage handle was opened as read only
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is not enough space in the underlying storage to save the value
- ESP_ERR_NVS_REMOVE_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

esp_err_t **nvs_set_u8** (*nvs_handle_t* handle, const char *key, uint8_t value)

set uint8_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_i16** (*nvs_handle_t* handle, const char *key, int16_t value)

set int16_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_u16** (*nvs_handle_t* handle, const char *key, uint16_t value)

set uint16_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_i32** (*nvs_handle_t* handle, const char *key, int32_t value)

set int32_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_u32** (*nvs_handle_t* handle, const char *key, uint32_t value)

set uint32_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_i64** (*nvs_handle_t* handle, const char *key, int64_t value)

set int64_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_u64** (*nvs_handle_t* handle, const char *key, uint64_t value)

set uint64_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_str** (*nvs_handle_t* handle, const char *key, const char *value)

set string for given key

Set value for the key, given its name. Note that the actual storage will not be updated until `nvs_commit` is called.

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **value** –**[in]** The value to set. For strings, the maximum length (including null character) is 4000 bytes, if there is one complete page free for writing. This decreases, however, if the free space is fragmented.

Returns

- ESP_OK if value was set successfully
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if storage handle was opened as read only
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is not enough space in the underlying storage to save the value
- ESP_ERR_NVS_REMOVE_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- ESP_ERR_NVS_VALUE_TOO_LONG if the string value is too long

esp_err_t **nvs_get_i8** (*nvs_handle_t* handle, const char *key, int8_t *out_value)

get int8_t value for given key

These functions retrieve value for the key, given its name. If `key` does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, out_value is not modified.

`out_value` has to be a pointer to an already allocated variable of the given type.

```
// Example of using nvs_get_i32:
int32_t max_buffer_size = 4096; // default value
esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
// if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
// have its default value.
```

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **out_value** –Pointer to the output value. May be NULL for nvs_get_str and nvs_get_blob, in this case required length will be returned in length argument.

Returns

- ESP_OK if the value was retrieved successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_INVALID_LENGTH if length is not sufficient to store data

esp_err_t **nvs_get_u8** (*nvs_handle_t* handle, const char *key, uint8_t *out_value)

get uint8_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_i16** (*nvs_handle_t* handle, const char *key, int16_t *out_value)

get int16_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_u16** (*nvs_handle_t* handle, const char *key, uint16_t *out_value)

get uint16_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_i32** (*nvs_handle_t* handle, const char *key, int32_t *out_value)

get int32_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_u32** (*nvs_handle_t* handle, const char *key, uint32_t *out_value)

get uint32_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_i64** (*nvs_handle_t* handle, const char *key, int64_t *out_value)

get int64_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_u64** (*nvs_handle_t* handle, const char *key, uint64_t *out_value)

get uint64_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

esp_err_t **nvs_get_str** (*nvs_handle_t* handle, const char *key, char *out_value, size_t *length)

get string value for given key

These functions retrieve the data of an entry, given its key. If key does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, out_value is not modified.

All functions expect out_value to be a pointer to an already allocated variable of the given type.

nvs_get_str and nvs_get_blob functions support WinAPI-style length queries. To get the size necessary to store the value, call nvs_get_str or nvs_get_blob with zero out_value and non-zero pointer to length. Variable pointed to by length argument will be set to the required length. For nvs_get_str, this length includes the zero terminator. When calling nvs_get_str and nvs_get_blob with non-zero out_value, length has to be non-zero and has to point to the length available in out_value. It is suggested that nvs_get/set_str is used for zero-terminated C strings, and nvs_get/set_blob used for arbitrary data structures.

```
// Example (without error checking) of using nvs_get_str to get a string into␣
↪dynamic array:
size_t required_size;
nvs_get_str(my_handle, "server_name", NULL, &required_size);
char* server_name = malloc(required_size);
nvs_get_str(my_handle, "server_name", server_name, &required_size);

// Example (without error checking) of using nvs_get_blob to get a binary data
into a static array:
uint8_t mac_addr[6];
size_t size = sizeof(mac_addr);
nvs_get_blob(my_handle, "dst_mac_addr", mac_addr, &size);
```

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **out_value** –**[out]** Pointer to the output value. May be NULL for nvs_get_str and nvs_get_blob, in this case required length will be returned in length argument.
- **length** –**[inout]** A non-zero pointer to the variable holding the length of out_value. In case out_value a zero, will be set to the length required to hold the value. In case out_value is not zero, will be set to the actual length of the value written. For nvs_get_str this includes zero terminator.

Returns

- ESP_OK if the value was retrieved successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_INVALID_LENGTH if `length` is not sufficient to store data

esp_err_t **nvs_get_blob**(*nvs_handle_t* handle, const char *key, void *out_value, size_t *length)

get blob value for given key

This function behaves the same as `nvs_get_str`, except for the data type.

esp_err_t **nvs_open**(const char *namespace_name, *nvs_open_mode_t* open_mode, *nvs_handle_t* *out_handle)

Open non-volatile storage with a given namespace from the default NVS partition.

Multiple internal ESP-IDF and third party application modules can store their key-value pairs in the NVS module. In order to reduce possible conflicts on key names, each module can use its own namespace. The default NVS partition is the one that is labelled "nvs" in the partition table.

Parameters

- **namespace_name** –**[in]** Namespace name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **open_mode** –**[in]** NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- **out_handle** –**[out]** If successful (return code is zero), handle will be returned in this argument.

Returns

- ESP_OK if storage handle was opened successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with label "nvs" is not found
- ESP_ERR_NVS_NOT_FOUND id namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is no space for a new entry or there are too many different namespaces (maximum allowed different namespaces: 254)
- other error codes from the underlying storage driver

esp_err_t **nvs_open_from_partition**(const char *part_name, const char *namespace_name, *nvs_open_mode_t* open_mode, *nvs_handle_t* *out_handle)

Open non-volatile storage with a given namespace from specified partition.

The behaviour is same as nvs_open() API. However this API can operate on a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using nvs_flash_init_partition() API.

Parameters

- **part_name** –**[in]** Label (name) of the partition of interest for object read/write/erase
- **namespace_name** –**[in]** Namespace name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **open_mode** –**[in]** NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- **out_handle** –**[out]** If successful (return code is zero), handle will be returned in this argument.

Returns

- ESP_OK if storage handle was opened successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with specified name is not found
- ESP_ERR_NVS_NOT_FOUND id namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is no space for a new entry or there are too many different namespaces (maximum allowed different namespaces: 254)
- other error codes from the underlying storage driver

esp_err_t **nvs_set_blob** (*nvs_handle_t* handle, const char *key, const void *value, size_t length)

set variable length binary value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until nvs_commit function is called.

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- **value** –**[in]** The value to set.
- **length** –**[in]** length of binary value to set, in bytes; Maximum length is 508000 bytes or (97.6% of the partition size - 4000) bytes whichever is lower.

Returns

- ESP_OK if value was set successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if storage handle was opened as read only
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is not enough space in the underlying storage to save the value
- ESP_ERR_NVS_REMOVE_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- ESP_ERR_NVS_VALUE_TOO_LONG if the value is too long

esp_err_t **nvs_erase_key** (*nvs_handle_t* handle, const char *key)

Erase key-value pair with given key name.

Note that actual storage may not be updated until nvs_commit function is called.

Parameters

- **handle** –**[in]** Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.
- **key** –**[in]** Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.

Returns

- ESP_OK if erase operation was successful

- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if handle was opened as read only
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- other error codes from the underlying storage driver

esp_err_t **nvs_erase_all**(*nvs_handle_t* handle)

Erase all key-value pairs in a namespace.

Note that actual storage may not be updated until nvs_commit function is called.

Parameters **handle** – **[in]** Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.

Returns

- ESP_OK if erase operation was successful
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if handle was opened as read only
- other error codes from the underlying storage driver

esp_err_t **nvs_commit**(*nvs_handle_t* handle)

Write any pending changes to non-volatile storage.

After setting any values, nvs_commit() must be called to ensure changes are written to non-volatile storage. Individual implementations may write to storage at other times, but this is not guaranteed.

Parameters **handle** – **[in]** Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.

Returns

- ESP_OK if the changes have been written successfully
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- other error codes from the underlying storage driver

void **nvs_close**(*nvs_handle_t* handle)

Close the storage handle and free any allocated resources.

This function should be called for each handle opened with nvs_open once the handle is not in use any more. Closing the handle may not automatically write the changes to nonvolatile storage. This has to be done explicitly using nvs_commit function. Once this function is called on a handle, the handle should no longer be used.

Parameters **handle** – **[in]** Storage handle to close

esp_err_t **nvs_get_stats**(const char *part_name, *nvs_stats_t* *nvs_stats)

Fill structure *nvs_stats_t*. It provides info about used memory the partition.

This function calculates to runtime the number of used entries, free entries, total entries, and amount namespace in partition.

```
// Example of nvs_get_stats() to get the number of used entries and free␣
↪entries:
nvs_stats_t nvs_stats;
nvs_get_stats(NULL, &nvs_stats);
printf("Count: UsedEntries = (%d), FreeEntries = (%d), AllEntries = (%d)\n",
       nvs_stats.used_entries, nvs_stats.free_entries, nvs_stats.total_
↪entries);
```

Parameters

- **part_name** – **[in]** Partition name NVS in the partition table. If pass a NULL than will use NVS_DEFAULT_PART_NAME ("nvs").

- **nvs_stats** –**[out]** Returns filled structure nvs_states_t. It provides info about used memory the partition.

Returns

- ESP_OK if the changes have been written successfully. Return param nvs_stats will be filled.
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with label "name" is not found. Return param nvs_stats will be filled 0.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param nvs_stats will be filled 0.
- ESP_ERR_INVALID_ARG if nvs_stats equal to NULL.
- ESP_ERR_INVALID_STATE if there is page with the status of INVALID. Return param nvs_stats will be filled not with correct values because not all pages will be counted. Counting will be interrupted at the first INVALID page.

esp_err_t **nvs_get_used_entry_count** (*nvs_handle_t* handle, size_t *used_entries)

Calculate all entries in a namespace.

An entry represents the smallest storage unit in NVS. Strings and blobs may occupy more than one entry. Note that to find out the total number of entries occupied by the namespace, add one to the returned value used_entries (if err is equal to ESP_OK). Because the name space entry takes one entry.

```
// Example of nvs_get_used_entry_count() to get amount of all key-value pairs␣
↪in one namespace:
nvs_handle_t handle;
nvs_open("namespace1", NVS_READWRITE, &handle);
...
size_t used_entries;
size_t total_entries_namespace;
if(nvs_get_used_entry_count(handle, &used_entries) == ESP_OK){
    // the total number of entries occupied by the namespace
    total_entries_namespace = used_entries + 1;
}
```

Parameters

- **handle** –**[in]** Handle obtained from nvs_open function.
- **used_entries** –**[out]** Returns amount of used entries from a namespace.

Returns

- ESP_OK if the changes have been written successfully. Return param used_entries will be filled valid value.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param used_entries will be filled 0.
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL. Return param used_entries will be filled 0.
- ESP_ERR_INVALID_ARG if used_entries equal to NULL.
- Other error codes from the underlying storage driver. Return param used_entries will be filled 0.

esp_err_t **nvs_entry_find** (const char *part_name, const char *namespace_name, *nvs_type_t* type, *nvs_iterator_t* *output_iterator)

Create an iterator to enumerate NVS entries based on one or more parameters.

```
// Example of listing all the key-value pairs of any type under specified␣
↪partition and namespace
 nvs_iterator_t it = NULL;
 esp_err_t res = nvs_entry_find(<nvs_partition_name>, <namespace>, NVS_TYPE_
↪ANY, &it);
```

(continues on next page)

(continued from previous page)

```
while(res == ESP_OK) {
    nvs_entry_info_t info;
    nvs_entry_info(it, &info); // Can omit error check if parameters are␣
↪guaranteed to be non-NULL
    printf("key '%s', type '%d' \n", info.key, info.type);
    res = nvs_entry_next(&it);
}
nvs_release_iterator(it);
```

Parameters

- **part_name** –**[in]** Partition name
- **namespace_name** –**[in]** Set this value if looking for entries with a specific namespace. Pass NULL otherwise.
- **type** –**[in]** One of nvs_type_t values.
- **output_iterator** –**[out]** Set to a valid iterator to enumerate all the entries found. Set to NULL if no entry for specified criteria was found. If any other error except ESP_ERR_INVALID_ARG occurs, `output_iterator` is NULL, too. If ESP_ERR_INVALID_ARG occurs, `output_iterator` is not changed. If a valid iterator is obtained through this function, it has to be released using `nvs_release_iterator` when not used any more, unless ESP_ERR_INVALID_ARG is returned.

Returns

- ESP_OK if no internal error or programming error occurred.
- ESP_ERR_NVS_NOT_FOUND if no element of specified criteria has been found.
- ESP_ERR_NO_MEM if memory has been exhausted during allocation of internal structures.
- ESP_ERR_INVALID_ARG if any of the parameters is NULL. Note: don't release `output_iterator` in case ESP_ERR_INVALID_ARG has been returned

esp_err_t **nvs_entry_next**(*nvs_iterator_t* *iterator)

Advances the iterator to next item matching the iterator criteria.

Note that any copies of the iterator will be invalid after this call.

Parameters **iterator** –**[inout]** Iterator obtained from nvs_entry_find function. Must be non-NULL. If any error except ESP_ERR_INVALID_ARG occurs, `iterator` is set to NULL. If ESP_ERR_INVALID_ARG occurs, `iterator` is not changed.

Returns

- ESP_OK if no internal error or programming error occurred.
- ESP_ERR_NVS_NOT_FOUND if no next element matching the iterator criteria.
- ESP_ERR_INVALID_ARG if `iterator` is NULL.
- Possibly other errors in the future for internal programming or flash errors.

esp_err_t **nvs_entry_info**(const *nvs_iterator_t* iterator, *nvs_entry_info_t* *out_info)

Fills *nvs_entry_info_t* structure with information about entry pointed to by the iterator.

Parameters

- **iterator** –**[in]** Iterator obtained from nvs_entry_find function. Must be non-NULL.
- **out_info** –**[out]** Structure to which entry information is copied.

Returns

- ESP_OK if all parameters are valid; current iterator data has been written to out_info
- ESP_ERR_INVALID_ARG if one of the parameters is NULL.

void **nvs_release_iterator**(*nvs_iterator_t* iterator)

Release iterator.

Parameters **iterator** –**[in]** Release iterator obtained from nvs_entry_find function. NULL argument is allowed.

Structures

struct **nvs_entry_info_t**

information about entry obtained from nvs_entry_info function

Public Members

char **namespace_name**[16]

Namespace to which key-value belong

char **key**[NVS_KEY_NAME_MAX_SIZE]

Key of stored key-value pair

nvs_type_t **type**

Type of stored key-value pair

struct **nvs_stats_t**

Note: Info about storage space NVS.

Public Members

size_t **used_entries**

Amount of used entries.

size_t **free_entries**

Amount of free entries.

size_t **total_entries**

Amount all available entries.

size_t **namespace_count**

Amount name space.

Macros

ESP_ERR_NVS_BASE

Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED

The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND

A requested entry couldn't be found or namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH

The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY

Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE

There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME

Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE

Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED

The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG

Key name is too long

ESP_ERR_NVS_PAGE_FULL

Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE

NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH

String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES

NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG

Value doesn't fit into the entry or string or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND

Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND

NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED

XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED

XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED

XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND

XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED

NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED

NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART

NVS key partition is corrupt

ESP_ERR_NVS_WRONG_ENCRYPTION

NVS partition is marked as encrypted with generic flash encryption. This is forbidden since the NVS encryption works differently.

ESP_ERR_NVS_CONTENT_DIFFERS

Internal error; never returned by nvs API functions. NVS key is different in comparison

NVS_DEFAULT_PART_NAME

Default partition name of the NVS partition in the partition table

NVS_PART_NAME_MAX_SIZE

maximum length of partition name (excluding null terminator)

NVS_KEY_NAME_MAX_SIZE

Maximum length of NVS key name (including null terminator)

Type Definitions

typedef uint32_t **nvs_handle_t**

Opaque pointer type representing non-volatile storage handle

typedef *nvs_handle_t* **nvs_handle**

typedef *nvs_open_mode_t* **nvs_open_mode**

typedef struct nvs_opaque_iterator_t ***nvs_iterator_t**

Opaque pointer type representing iterator to nvs entries

Enumerations

enum **nvs_open_mode_t**

Mode of opening the non-volatile storage.

Values:

enumerator **NVS_READONLY**

Read only

enumerator **NVS_READWRITE**

Read and write

enum **nvs_type_t**

Types of variables.

Values:

enumerator **NVS_TYPE_U8**

Type uint8_t

enumerator **NVS_TYPE_I8**

Type int8_t

enumerator **NVS_TYPE_U16**

Type uint16_t

enumerator **NVS_TYPE_I16**

Type int16_t

enumerator **NVS_TYPE_U32**

Type uint32_t

enumerator **NVS_TYPE_I32**

Type int32_t

enumerator **NVS_TYPE_U64**

Type uint64_t

enumerator **NVS_TYPE_I64**

Type int64_t

enumerator **NVS_TYPE_STR**

Type string

enumerator **NVS_TYPE_BLOB**

Type blob

enumerator **NVS_TYPE_ANY**

Must be last

2.9.4 NVS Partition Generator Utility

Introduction

The utility nvs_flash/nvs_partition_generator/nvs_partition_gen.py creates a binary file based on key-value pairs provided in a CSV file. The binary file is compatible with NVS architecture defined in *Non-Volatile Storage*. This utility is ideally suited for generating a binary blob, containing data specific to ODM/OEM, which can be flashed externally

at the time of device manufacturing. This allows manufacturers to generate many instances of the same application firmware with customized parameters for each device, such as a serial number.

Prerequisites

To use this utility in encryption mode, install the following packages:

- cryptography package

All the required packages are included in *requirements.txt* in the root of the esp-idf directory.

CSV File Format

Each line of a CSV file should contain 4 parameters, separated by a comma. The table below provides the description for each of these parameters.

No.	Parameter	Description	Notes
1	Key	Key of the data. The data can be accessed later from an application using this key.	
2	Type	Supported values are `file`, `data`, and `namespace`.	
3	Encoding	Supported values are: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `string`, `hex2bin`, `base64`, and `binary`. This specifies how actual data values are encoded in the resulting binary file. The difference between the `string` and `binary` encoding is that `string` data is terminated with a NULL character, whereas `binary` data is not.	As of now, for the `file` type, only `hex2bin`, `base64`, `string`, and `binary` encoding is supported.
4	Value	Data value	`Encoding` and `Value` cells for the `namespace` field type should be empty. `Encoding` and `Value` of `namespace` are fixed and are not configurable. Any values in these cells are ignored.

Note: The first line of the CSV file should always be the column header and it is not configurable.

Below is an example dump of such a CSV file:

```
key,type,encoding,value      <-- column header
namespace_name,namespace,,   <-- First entry should be of type "namespace"
key1,data,u8,1
key2,file,string,/path/to/file
```

Note:

Make sure there are no spaces:

- before and after ‘,’
- at the end of each line in a CSV file

NVS Entry and Namespace Association

When a namespace entry is encountered in a CSV file, each following entry will be treated as part of that namespace until the next namespace entry is found. At this point, all the following entries will be treated as part of the new namespace.

Note: First entry in a CSV file should always be a `namespace` entry.

Multipage Blob Support

By default, binary blobs are allowed to span over multiple pages and are written in the format mentioned in Section *Structure of Entry*. If you intend to use an older format, the utility provides an option to disable this feature.

Encryption Support

The NVS Partition Generator utility also allows you to create an encrypted binary file. The utility uses the AES-XTS encryption. Please refer to *NVS Encryption* for more details.

Decryption Support

This utility allows you to decrypt an encrypted NVS binary file. The utility uses an NVS binary file encrypted using AES-XTS encryption. Please refer to *NVS Encryption* for more details.

Running the Utility

Usage:

```
python nvs_partition_gen.py [-h] {generate,generate-key,encrypt,decrypt} ...
```

Optional Arguments:

No.	Parameter	Description
1	-h, –help	Show this help message and exit

Commands:

```
Run nvs_partition_gen.py {command} -h for additional help
```

No.	Parameter	Description
1	generate	Generate NVS partition
2	generate-key	Generate keys for encryption
3	encrypt	Generate NVS encrypted partition
4	decrypt	Decrypt NVS encrypted partition

To Generate NVS Partition (Default): **Usage**:

```
python nvs_partition_gen.py generate [-h] [--version {1,2}] [--outdir OUTDIR]
                                     input output size
```

Positional Arguments:

Parameter	Description
input	Path to CSV file to parse
output	Path to output NVS binary file
size	Size of NVS partition in bytes (must be multiple of 4096)

Optional Arguments:

Parameter	Description
-h, –help	Show this help message and exit
–version {1,2}	Set multipage blob version Version 1 - Multipage blob support disabled Version 2 - Multipage blob support enabled Default: Version 2
–outdir OUTDIR	Output directory to store files created (Default: current directory)

You can run the utility to generate NVS partition using the command below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_singlepage_blob.csv sample.bin 0x3000
```

To Generate Only Encryption Key Partition: **Usage**:

```
python nvs_partition_gen.py generate-key [-h] [--keyfile KEYFILE]
                                         [--outdir OUTDIR]
```

Optional Arguments:

Parameter	Description
-h, –help	Show this help message and exit
–keyfile KEYFILE	Path to output encryption key partition file
–outdir OUTDIR	Output directory to store file created (Default: current directory)

You can run the utility to generate only the encryption key partition using the command below:

```
python nvs_partition_gen.py generate-key
```

To Generate Encrypted NVS Partition: **Usage**:

```
python nvs_partition_gen.py encrypt [-h] [--version {1,2}] [--keygen]
                                    [--keyfile KEYFILE] [--inputkey INPUTKEY]
                                    [--outdir OUTDIR]
                                    input output size
```

Positional Arguments:

Parameter	Description
input	Path to CSV file to parse
output	Path to output NVS binary file
size	Size of NVS partition in bytes (must be multiple of 4096)

Optional Arguments:

Parameter	Description
-h, –help	Show this help message and exit
–version {1,2}	Set multipage blob version Version 1 - Multipage blob support disabled Version 2 - Multipage blob support enabled Default: Version 2
–keygen	Generates key for encrypting NVS partition
–keyfile KEYFILE	Path to output encryption keys file
–inputkey INPUTKEY	File having key for encrypting NVS partition
–outdir OUTDIR	Output directory to store files created (Default: current directory)

You can run the utility to encrypt NVS partition using the command below. A sample CSV file is provided with the utility:

- Encrypt by allowing the utility to generate encryption keys:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin␣
↪0x3000 --keygen
```

Note: Encryption key of the following format `<outdir>/keys/keys-<timestamp>.bin` is created.

- Encrypt by allowing the utility to generate encryption keys and store it in provided custom filename:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin␣
↪0x3000 --keygen --keyfile sample_keys.bin
```

Note: Encryption key of the following format `<outdir>/keys/sample_keys.bin` is created.

Note: This newly created file having encryption keys in `keys/` directory is compatible with NVS key-partition structure. Refer to *NVS Key Partition* for more details.

- Encrypt by providing the encryption keys as input binary file:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin␣
↪0x3000 --inputkey sample_keys.bin
```

To Decrypt Encrypted NVS Partition: **Usage**:

```
python nvs_partition_gen.py decrypt [-h] [--outdir OUTDIR] input key output
```

Positional Arguments:

Parameter	Description
input	Path to encrypted NVS partition file to parse
key	Path to file having keys for decryption
output	Path to output decrypted binary file

Optional Arguments:

Parameter	Description
-h, –help	Show this help message and exit
–outdir OUTDIR	Output directory to store files created (Default: current directory)

You can run the utility to decrypt encrypted NVS partition using the command below:

```
python nvs_partition_gen.py decrypt sample_encr.bin sample_keys.bin sample_decr.bin
```

You can also provide the format version number:

- Multipage Blob Support Disabled (Version 1)
- Multipage Blob Support Enabled (Version 2)

Multipage Blob Support Disabled (Version 1): You can run the utility in this format by setting the version parameter to 1, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_singlepage_blob.csv sample.bin 0x3000 --
↪version 1
```

Multipage Blob Support Enabled (Version 2): You can run the utility in this format by setting the version parameter to 2, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_multipage_blob.csv sample.bin 0x4000 --
↪version 2
```

Note: *Minimum NVS Partition Size needed is 0x3000 bytes.*

Note: *When flashing the binary onto the device, make sure it is consistent with the application's sdkconfig.*

Caveats

- Utility does not check for duplicate keys and will write data pertaining to both keys. You need to make sure that the keys are distinct.
- Once a new page is created, no data will be written in the space left on the previous page. Fields in the CSV file need to be ordered in such a way as to optimize memory.
- 64-bit datatype is not yet supported.

2.9.5 SD/SDIO/MMC Driver

Overview

The SD/SDIO/MMC driver currently supports SD memory, SDIO cards, and eMMC chips. This is a protocol level driver built on top of SDMMC and SD SPI host drivers.

SDMMC and SD SPI host drivers (driver/include/driver/sdmmc_host.h and driver/include/driver/sdspi_host.h) provide API functions for:

- Sending commands to slave devices
- Sending and receiving data
- Handling error conditions within the bus

For functions used to initialize and configure:

- SDMMC host, see *SDMMC Host API*
- SD SPI host, see *SD SPI Host API*

The SDMMC protocol layer described in this document handles the specifics of the SD protocol, such as the card initialization and data transfer commands.

The protocol layer works with the host via the *sdmmc_host_t* structure. This structure contains pointers to various functions of the host.

Application Example

An example which combines the SDMMC driver with the FATFS library is provided in the storage/sd_card directory of ESP-IDF examples. This example initializes the card, then writes and reads data from it using POSIX and C library APIs. See README.md file in the example directory for more information.

Combo (memory + IO) cards The driver does not support SD combo cards. Combo cards are treated as IO cards.

Thread safety Most applications need to use the protocol layer only in one task. For this reason, the protocol layer does not implement any kind of locking on the *sdmmc_card_t* structure, or when accessing SDMMC or SD SPI host drivers. Such locking is usually implemented on a higher layer, e.g., in the filesystem driver.

Protocol layer API

The protocol layer is given the *sdmmc_host_t* structure. This structure describes the SD/MMC host driver, lists its capabilities, and provides pointers to functions of the driver. The protocol layer stores card-specific information in the *sdmmc_card_t* structure. When sending commands to the SD/MMC host driver, the protocol layer uses the *sdmmc_command_t* structure to describe the command, arguments, expected return values, and data to transfer if there is any.

Using API with SD memory cards

1. To initialize the host, call the host driver functions, e.g., *sdmmc_host_init()*, *sdmmc_host_init_slot()*.
2. To initialize the card, call *sdmmc_card_init()* and pass to it the parameters `host` - the host driver information, and `card` - a pointer to the structure *sdmmc_card_t* which will be filled with information about the card when the function completes.
3. To read and write sectors of the card, use *sdmmc_read_sectors()* and *sdmmc_write_sectors()* respectively and pass to it the parameter `card` - a pointer to the card information structure.
4. If the card is not used anymore, call the host driver function - e.g., *sdmmc_host_deinit()* - to disable the host peripheral and free the resources allocated by the driver.

Using API with eMMC chips From the protocol layer's perspective, eMMC memory chips behave exactly like SD memory cards. Even though eMMCs are chips and do not have a card form factor, the terminology for SD cards can still be applied to eMMC due to the similarity of the protocol (*sdmmc_card_t*, *sdmmc_card_init*). Note that eMMC chips cannot be used over SPI, which makes them incompatible with the SD SPI host driver.

To initialize eMMC memory and perform read/write operations, follow the steps listed for SD cards in the previous section.

Using API with SDIO cards Initialization and the probing process are the same as with SD memory cards. The only difference is in data transfer commands in SDIO mode.

During the card initialization and probing, performed with *sdmmc_card_init()*, the driver only configures the following registers of the IO card:

1. The IO portion of the card is reset by setting RES bit in the I/O Abort (0x06) register.
2. If 4-line mode is enabled in host and slot configuration, the driver attempts to set the Bus width field in the Bus Interface Control (0x07) register. If setting the filed is successful, which means that the slave supports 4-line mode, the host is also switched to 4-line mode.

3. If high-speed mode is enabled in the host configuration, the SHS bit is set in the High Speed (0x13) register.

In particular, the driver does not set any bits in (1) I/O Enable and Int Enable registers, (2) I/O block sizes, etc. Applications can set them by calling *sdmmc_io_write_byte()*.

For card configuration and data transfer, choose the pair of functions relevant to your case from the table below.

Action	Read Function	Write Function
Read and write a single byte using IO_RW_DIRECT (CMD52)	*sdmmc_io_read_byte()*	*sdmmc_io_write_byte()*
Read and write multiple bytes using IO_RW_EXTENDED (CMD53) in byte mode	*sdmmc_io_read_bytes()*	*sdmmc_io_write_bytes()*
Read and write blocks of data using IO_RW_EXTENDED (CMD53) in block mode	*sdmmc_io_read_blocks()*	*sdmmc_io_write_blocks()*

SDIO interrupts can be enabled by the application using the function *sdmmc_io_enable_int()*. When using SDIO in 1-line mode, the D1 line also needs to be connected to use SDIO interrupts.

If you want the application to wait until the SDIO interrupt occurs, use *sdmmc_io_wait_int()*.

There is a component ESSL (ESP Serial Slave Link) to use if you are communicating with an ESP32 SDIO slave. See *ESP Serial Slave Link* and example peripherals/sdio/host.

API Reference

Header File

- components/sdmmc/include/sdmmc_cmd.h

Functions

esp_err_t **sdmmc_card_init**(const *sdmmc_host_t* *host, *sdmmc_card_t* *out_card)

Probe and initialize SD/MMC card using given host

Note: Only SD cards (SDSC and SDHC/SDXC) are supported now. Support for MMC/eMMC cards will be added later.

Parameters
- **host** –pointer to structure defining host controller
- **out_card** –pointer to structure which will receive information about the card when the function completes

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

void **sdmmc_card_print_info**(FILE *stream, const *sdmmc_card_t* *card)

Print information about the card to a stream.

Parameters
- **stream** –stream obtained using fopen or fdopen
- **card** –card information structure initialized using sdmmc_card_init

esp_err_t **sdmmc_get_status**(*sdmmc_card_t* *card)

Get status of SD/MMC card

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns
- ESP_OK on success

- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_write_sectors**(*sdmmc_card_t* *card, const void *src, size_t start_sector, size_t sector_count)

Write given number of sectors to SD/MMC card

Parameters
- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **src** –pointer to data buffer to read data from; data size must be equal to sector_count * card->csd.sector_size
- **start_sector** –sector where to start writing
- **sector_count** –number of sectors to write

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_read_sectors**(*sdmmc_card_t* *card, void *dst, size_t start_sector, size_t sector_count)

Read given number of sectors from the SD/MMC card

Parameters
- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **dst** –pointer to data buffer to write into; buffer size must be at least sector_count * card->csd.sector_size
- **start_sector** –sector where to start reading
- **sector_count** –number of sectors to read

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_erase_sectors**(*sdmmc_card_t* *card, size_t start_sector, size_t sector_count, *sdmmc_erase_arg_t* arg)

Erase given number of sectors from the SD/MMC card

Note: When sdmmc_erase_sectors used with cards in SDSPI mode, it was observed that card requires re-init after erase operation.

Parameters
- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **start_sector** –sector where to start erase
- **sector_count** –number of sectors to erase
- **arg** –erase command (CMD38) argument

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_can_discard**(*sdmmc_card_t* *card)

Check if SD/MMC card supports discard

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns
- ESP_OK if supported by the card/device
- ESP_FAIL if not supported by the card/device

esp_err_t **sdmmc_can_trim**(*sdmmc_card_t* *card)

Check if SD/MMC card supports trim

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns

- ESP_OK if supported by the card/device
- ESP_FAIL if not supported by the card/device

esp_err_t **sdmmc_mmc_can_sanitize**(*sdmmc_card_t* *card)

Check if SD/MMC card supports sanitize

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns

- ESP_OK if supported by the card/device
- ESP_FAIL if not supported by the card/device

esp_err_t **sdmmc_mmc_sanitize**(*sdmmc_card_t* *card, uint32_t timeout_ms)

Sanitize the data that was unmapped by a Discard command

Note: Discard command has to precede sanitize operation. To discard, use MMC_DICARD_ARG with sdmmc_erase_sectors argument

---

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **timeout_ms** –timeout value in milliseconds required to sanitize the selected range of sectors.

Returns

- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_full_erase**(*sdmmc_card_t* *card)

Erase complete SD/MMC card

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns

- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_read_byte**(*sdmmc_card_t* *card, uint32_t function, uint32_t reg, uint8_t *out_byte)

Read one byte from an SDIO card using IO_RW_DIRECT (CMD52)

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **function** –IO function number
- **reg** –byte address within IO function
- **out_byte** –**[out]** output, receives the value read from the card

Returns

- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_write_byte**(*sdmmc_card_t* *card, uint32_t function, uint32_t reg, uint8_t in_byte, uint8_t *out_byte)

Write one byte to an SDIO card using IO_RW_DIRECT (CMD52)

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **function** –IO function number
- **reg** –byte address within IO function
- **in_byte** –value to be written
- **out_byte** –**[out]** if not NULL, receives new byte value read from the card (read-after-write).

Returns

- ESP_OK on success
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_read_bytes**(*sdmmc_card_t* *card, uint32_t function, uint32_t addr, void *dst, size_t size)

Read multiple bytes from an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs read operation using CMD53 in byte mode. For block mode, see sdmmc_io_read_blocks.

Parameters

- **card** – pointer to card information structure previously initialized using sdmmc_card_init
- **function** – IO function number
- **addr** – byte address within IO function where reading starts
- **dst** – buffer which receives the data read from card
- **size** – number of bytes to read

Returns

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_write_bytes**(*sdmmc_card_t* *card, uint32_t function, uint32_t addr, const void *src, size_t size)

Write multiple bytes to an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs write operation using CMD53 in byte mode. For block mode, see sdmmc_io_write_blocks.

Parameters

- **card** – pointer to card information structure previously initialized using sdmmc_card_init
- **function** – IO function number
- **addr** – byte address within IO function where writing starts
- **src** – data to be written
- **size** – number of bytes to write

Returns

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_read_blocks**(*sdmmc_card_t* *card, uint32_t function, uint32_t addr, void *dst, size_t size)

Read blocks of data from an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs read operation using CMD53 in block mode. For byte mode, see sdmmc_io_read_bytes.

Parameters

- **card** – pointer to card information structure previously initialized using sdmmc_card_init
- **function** – IO function number
- **addr** – byte address within IO function where writing starts
- **dst** – buffer which receives the data read from card
- **size** – number of bytes to read, must be divisible by the card block size.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_write_blocks**(*sdmmc_card_t* *card, uint32_t function, uint32_t addr, const void *src, size_t size)

Write blocks of data to an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs write operation using CMD53 in block mode. For byte mode, see sdmmc_io_write_bytes.

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **function** –IO function number
- **addr** –byte address within IO function where writing starts
- **src** –data to be written
- **size** –number of bytes to read, must be divisible by the card block size.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

esp_err_t **sdmmc_io_enable_int**(*sdmmc_card_t* *card)

Enable SDIO interrupt in the SDMMC host

Parameters **card** –pointer to card information structure previously initialized using sdmmc_card_init

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts

esp_err_t **sdmmc_io_wait_int**(*sdmmc_card_t* *card, TickType_t timeout_ticks)

Block until an SDIO interrupt is received

Slave uses D1 line to signal interrupt condition to the host. This function can be used to wait for the interrupt.

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **timeout_ticks** –time to wait for the interrupt, in RTOS ticks

Returns

- ESP_OK if the interrupt is received
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts
- ESP_ERR_TIMEOUT if the interrupt does not happen in timeout_ticks

esp_err_t **sdmmc_io_get_cis_data**(*sdmmc_card_t* *card, uint8_t *out_buffer, size_t buffer_size, size_t *inout_cis_size)

Get the data of CIS region of an SDIO card.

You may provide a buffer not sufficient to store all the CIS data. In this case, this function stores as much data into your buffer as possible. Also, this function will try to get and return the size required for you.

Parameters

- **card** –pointer to card information structure previously initialized using sdmmc_card_init
- **out_buffer** –Output buffer of the CIS data
- **buffer_size** –Size of the buffer.
- **inout_cis_size** –Mandatory, pointer to a size, input and output.
  - input: Limitation of maximum searching range, should be 0 or larger than buffer_size. The function searches for CIS_CODE_END until this range. Set to 0 to search infinitely.
  - output: The size required to store all the CIS data, if CIS_CODE_END is found.

Returns

- ESP_OK: on success
- ESP_ERR_INVALID_RESPONSE: if the card does not (correctly) support CIS.
- ESP_ERR_INVALID_SIZE: CIS_CODE_END found, but buffer_size is less than required size, which is stored in the inout_cis_size then.
- ESP_ERR_NOT_FOUND: if the CIS_CODE_END not found. Increase input value of inout_cis_size or set it to 0, if you still want to search for the end; output value of inout_cis_size is invalid in this case.
- and other error code return from sdmmc_io_read_bytes

esp_err_t **sdmmc_io_print_cis_info**(uint8_t *buffer, size_t buffer_size, FILE *fp)

Parse and print the CIS information of an SDIO card.

Note: Not all the CIS codes and all kinds of tuples are supported. If you see some unresolved code, you can add the parsing of these code in sdmmc_io.c and contribute to the IDF through the Github repository.

```
using sdmmc_card_init
```

Parameters

- **buffer** –Buffer to parse
- **buffer_size** –Size of the buffer.
- **fp** –File pointer to print to, set to NULL to print to stdout.

Returns

- ESP_OK: on success
- ESP_ERR_NOT_SUPPORTED: if the value from the card is not supported to be parsed.
- ESP_ERR_INVALID_SIZE: if the CIS size fields are not correct.

Header File

- components/driver/include/driver/sdmmc_types.h

Structures

struct **sdmmc_csd_t**

Decoded values from SD card Card Specific Data register

Public Members

int **csd_ver**

CSD structure format

int **mmc_ver**

MMC version (for CID format)

int **capacity**

total number of sectors

int **sector_size**

sector size in bytes

int **read_block_len**

block length for reads

int **card_command_class**

Card Command Class for SD

int **tr_speed**

Max transfer speed

struct **sdmmc_cid_t**

Decoded values from SD card Card IDentification register

Public Members

int **mfg_id**
: manufacturer identification number

int **oem_id**
: OEM/product identification number

char **name**[8]
: product name (MMC v1 has the longest)

int **revision**
: product revision

int **serial**
: product serial number

int **date**
: manufacturing date

struct **sdmmc_scr_t**

Decoded values from SD Configuration Register Note: When new member is added, update reserved bits accordingly

Public Members

uint32_t **sd_spec**
: SD Physical layer specification version, reported by card

uint32_t **erase_mem_state**
: data state on card after erase whether 0 or 1 (card vendor dependent)

uint32_t **bus_width**
: bus widths supported by card: BIT(0) —1-bit bus, BIT(2) —4-bit bus

uint32_t **reserved**
: reserved for future expansion

uint32_t **rsvd_mnf**
: reserved for manufacturer usage

struct **sdmmc_ssr_t**

Decoded values from SD Status Register Note: When new member is added, update reserved bits accordingly

Public Members

uint32_t **alloc_unit_kb**
: Allocation unit of the card, in multiples of kB (1024 bytes)

uint32_t **erase_size_au**

Erase size for the purpose of timeout calculation, in multiples of allocation unit

uint32_t **cur_bus_width**

SD current bus width

uint32_t **discard_support**

SD discard feature support

uint32_t **fule_support**

SD FULE (Full User Area Logical Erase) feature support

uint32_t **erase_timeout**

Timeout (in seconds) for erase of a single allocation unit

uint32_t **erase_offset**

Constant timeout offset (in seconds) for any erase operation

uint32_t **reserved**

reserved for future expansion

struct **sdmmc_ext_csd_t**

Decoded values of Extended Card Specific Data

Public Members

uint8_t **rev**

Extended CSD Revision

uint8_t **power_class**

Power class used by the card

uint8_t **erase_mem_state**

data state on card after erase whether 0 or 1 (card vendor dependent)

uint8_t **sec_feature**

secure data management features supported by the card

struct **sdmmc_switch_func_rsp_t**

SD SWITCH_FUNC response buffer

Public Members

uint32_t **data**[512 / 8 / sizeof(uint32_t)]

response data

struct **sdmmc_command_t**

SD/MMC command information

Public Members

uint32_t **opcode**

SD or MMC command index

uint32_t **arg**

SD/MMC command argument

sdmmc_response_t **response**

response buffer

void ***data**

buffer to send or read into

size_t **datalen**

length of data buffer

size_t **blklen**

block length

int **flags**

see below

esp_err_t **error**

error returned from transfer

uint32_t **timeout_ms**

response timeout, in milliseconds

struct **sdmmc_host_t**

SD/MMC Host description

This structure defines properties of SD/MMC host and functions of SD/MMC host which can be used by upper layers.

Public Members

uint32_t **flags**

flags defining host properties

int **slot**

slot number, to be passed to host functions

int **max_freq_khz**

max frequency supported by the host

float **io_voltage**

I/O voltage used by the controller (voltage switching is not supported)

esp_err_t (*__init__)(void)

Host function to initialize the driver

esp_err_t (*__set_bus_width__)(int slot, size_t width)

host function to set bus width

size_t (*__get_bus_width__)(int slot)

host function to get bus width

esp_err_t (*__set_bus_ddr_mode__)(int slot, bool ddr_enable)

host function to set DDR mode

esp_err_t (*__set_card_clk__)(int slot, uint32_t freq_khz)

host function to set card clock frequency

esp_err_t (*__set_cclk_always_on__)(int slot, bool cclk_always_on)

host function to set whether the clock is always enabled

esp_err_t (*__do_transaction__)(int slot, *sdmmc_command_t* *cmdinfo)

host function to do a transaction

esp_err_t (*__deinit__)(void)

host function to deinitialize the driver

esp_err_t (*__deinit_p__)(int slot)

host function to deinitialize the driver, called with the `slot`

esp_err_t (*__io_int_enable__)(int slot)

Host function to enable SDIO interrupt line

esp_err_t (*__io_int_wait__)(int slot, TickType_t timeout_ticks)

Host function to wait for SDIO interrupt line to be active

int __command_timeout_ms__

timeout, in milliseconds, of a single command. Set to 0 to use the default value.

struct __sdmmc_card_t__

SD/MMC card information structure

Public Members

sdmmc_host_t __host__

Host with which the card is associated

uint32_t __ocr__

OCR (Operation Conditions Register) value

sdmmc_cid_t **cid**

decoded CID (Card IDentification) register value

sdmmc_response_t **raw_cid**

raw CID of MMC card to be decoded after the CSD is fetched in the data transfer mode

sdmmc_csd_t **csd**

decoded CSD (Card-Specific Data) register value

sdmmc_scr_t **scr**

decoded SCR (SD card Configuration Register) value

sdmmc_ssr_t **ssr**

decoded SSR (SD Status Register) value

sdmmc_ext_csd_t **ext_csd**

decoded EXT_CSD (Extended Card Specific Data) register value

uint16_t **rca**

RCA (Relative Card Address)

uint16_t **max_freq_khz**

Maximum frequency, in kHz, supported by the card

uint32_t **is_mem**

Bit indicates if the card is a memory card

uint32_t **is_sdio**

Bit indicates if the card is an IO card

uint32_t **is_mmc**

Bit indicates if the card is MMC

uint32_t **num_io_functions**

If is_sdio is 1, contains the number of IO functions on the card

uint32_t **log_bus_width**

log2(bus width supported by card)

uint32_t **is_ddr**

Card supports DDR mode

uint32_t **reserved**

Reserved for future expansion

Macros

SDMMC_HOST_FLAG_1BIT

host supports 1-line SD and MMC protocol

SDMMC_HOST_FLAG_4BIT

host supports 4-line SD and MMC protocol

SDMMC_HOST_FLAG_8BIT

host supports 8-line MMC protocol

SDMMC_HOST_FLAG_SPI

host supports SPI protocol

SDMMC_HOST_FLAG_DDR

host supports DDR mode for SD/MMC

SDMMC_HOST_FLAG_DEINIT_ARG

host `deinit` function called with the slot argument

SDMMC_FREQ_DEFAULT

SD/MMC Default speed (limited by clock divider)

SDMMC_FREQ_HIGHSPEED

SD High speed (limited by clock divider)

SDMMC_FREQ_PROBING

SD/MMC probing speed

SDMMC_FREQ_52M

MMC 52MHz speed

SDMMC_FREQ_26M

MMC 26MHz speed

Type Definitions

typedef uint32_t **sdmmc_response_t**[4]

SD/MMC command response buffer

Enumerations

enum **sdmmc_erase_arg_t**

SD/MMC erase command(38) arguments SD: ERASE: Erase the write blocks, physical/hard erase.

DISCARD: Card may deallocate the discarded blocks partially or completely. After discard operation the previously written data may be partially or fully read by the host depending on card implementation.

MMC: ERASE: Does TRIM, applies erase operation to write blocks instead of Erase Group.

DISCARD: The Discard function allows the host to identify data that is no longer required so that the device can erase the data if necessary during background erase events. Applies to write blocks instead of Erase Group After discard operation, the original data may be remained partially or fully accessible to the host dependent on device.

Values:

enumerator **SDMMC_ERASE_ARG**

Erase operation on SD, Trim operation on MMC

enumerator **SDMMC_DISCARD_ARG**

Discard operation for SD/MMC

2.9.6 SPI Flash API

Overview

The spi_flash component contains API functions related to reading, writing, erasing, memory mapping for data in the external flash. The spi_flash component also has higher-level API functions which work with partitions defined in the *partition table*.

Different from the API before IDF v4.0, the functionality of *esp_flash_** APIs is not limited to the "main" SPI flash chip (the same SPI flash chip from which program runs). With different chip pointers, you can access external flash chips connected to not only SPI0/1 but also other SPI buses like SPI2.

Note: Instead of going through the cache connected to the SPI0 peripheral, most *esp_flash_** APIs go through other SPI peripherals like SPI1, SPI2, etc. This makes them able to access not only the main flash, but also external flash.

However, due to limitations of the cache, operations through the cache are limited to the main flash. The address range limitation for these operations are also on the cache side. The cache is not able to access external flash chips or address range above its capabilities. These cache operations include: mmap, encrypted read/write, executing code or access to variables in the flash.

Note: Flash APIs after ESP-IDF v4.0 are no longer *atomic*. If a write operation occurs during another on-going read operation, and the flash addresses of both operations overlap, the data returned from the read operation may contain both old data and new data (that was updated written by the write operation).

Note: Encrypted flash operations are only supported with the main flash chip (and not with other flash chips, that is on SPI1 with different CS, or on other SPI buses). Reading through cache is only supported on the main flash, which is determined by the HW.

Support for Features of Flash Chips

Quad/Dual Mode Chips Features of different flashes are implemented in different ways and thus need speical support. The fast/slow read and Dual mode (DOUT/DIO) of almost all 24-bits address flash chips are supported, because they don't need any vendor-specific commands.

Quad mode (QIO/QOUT) is supported on following chip types:

1. ISSI
2. GD
3. MXIC
4. FM
5. Winbond
6. XMC
7. BOYA

Optional Features

Optional features for flash Some features are not supported on all ESP chips and Flash chips. You can check the list below for more information.

- *Auto Suspend & Resume*
- *Flash unique ID*
- *High performance mode*
- *OPI flash support*
- *32-bit Address Flash Chips*

Note: When Flash optional features listed in this page are used, aside from the capability of ESP chips, and ESP-IDF verison you are using, you will also need to make sure these features are supported by flash chips used.

- If you are using an official Espressif modules/SiP. Some of the modules/SiPs always support the feature, in this case you can see these features listed in the datasheet. Otherwise please contact Espressif's business team to know if we can supply such products for you.
- If you are making your own modules with your own bought flash chips, and you need features listed above. Please contact your vendor if they support the those features, and make sure that the chips can be supplied continuously.

Attention: This document only shows that IDF code has supported the features of those flash chips. It's not a list of stable flash chips certified by Espressif. If you build your own hardware from flash chips with your own brought flash chips (even with flash listed in this page), you need to validate the reliability of flash chips yourself.

Auto Suspend & Resume This feature is only supported on ESP32-C3 for now.

The support for ESP32-S3, ESP32-C2 may be added in the future.

Flash unique ID This feature is supported on all Espressif chips.

Unique ID is not flash id, which means flash has 64-Bit unique ID for each device. The instruction to read the unique ID (4Bh) accesses a factory-set read-only 64-bit number that is unique to each flash device. This ID number helps you to recognize each single device. Not all flash vendors support this feature. If you try to read the unique ID on a chip which does not have this feature, the behavior is not determined. The support list is as follows.

List of Flash chips that support this feature:

1. ISSI
2. GD
3. TH
4. FM
5. Winbond
6. XMC
7. BOYA

High performance mode This featuer is only supported on ESP32-S3 for now.

The support for ESP32-S2, ESP32-C3 may be added in the future.

OPI flash support This feature is only supporetd on ESP32-S3 for now.

OPI flash means that the flash chip supports octal peripheral interface, which has octal I/O pins. Different octal flash has different configurations and different commands. Hence, it is necessary to carefully check the support list.

32-bit Address Flash Chips This feature is supported on all Espressif chips (with various restrictions to application).

Most NOR flash chips used by Espressif chips use 24-bits address, which can cover 16 MBytes memory. However, for larger memory (usually equal to or larger than 16 MBytes), flash uses a 32-bits address to address larger memory. Regretfully, 32-bits address chips have vendor-specific commands, so we need to support the chips one by one.

List of Flash chips that support this feature:

1. W25Q256
2. GD25Q256

Important: Over 16 MBytes space on flash mentioned above can be only used for `data saving`, like file system. If your data/instructions over 16 MBytes spaces need to be mapped to MMU (so as to be accessed by the CPU), please upgrade to ESP-IDF v5.2 and read the latest docs.

There are some features that are not supported by all flash chips, or not supported by all Espressif chips. These features include:

- 32-bit address flash - usually means that the flash has higher capacity (equal to or larger than 16 MB) that needs longer addresses.
- Flash unique ID - means that flash supports its unique 64-bits ID.

If you want to use these features, please ensure both ESP32 and ALL flash chips in your product support these features. For more details, refer to *Optional features for flash*.

You may also customise your own flash chip driver. See *Overriding Default Chip Drivers* for more details.

Warning: Customizing SPI Flash Chip Drivers is considered an "expert" feature. Users should only do so at their own risk. (See the notes below)

Overriding Default Chip Drivers During the SPI Flash driver's initialization (i.e., `esp_flash_init()`), there is a chip detection step during which the driver will iterate through a Default Chip Driver List and determine which chip driver can properly support the currently connected flash chip. The Default Chip Drivers are provided by the IDF, thus are updated in together with each IDF version. However IDF also allows users to customize their own chip drivers.

Users should note the following when customizing chip drivers:

1. You may need to rely on some non-public IDF functions, which have slight possibility to change between IDF versions. On the one hand, these changes may be useful bug fixes for your driver, on the other hand, they may also be breaking changes (i.e., breaks your code).
2. Some IDF bug fixes to other chip drivers will not be automatically applied to your own custom chip drivers.
3. If the protection of flash is not handled properly, there may be some random reliability issues.
4. If you update to a newer IDF version that has support for more chips, you will have to manually add those new chip drivers into your custom chip driver list. Otherwise the driver will only search for the drivers in custom list you provided.

Steps For Creating Custom Chip Drivers and Overriding the IDF Default Driver List

1. Enable the *CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST* config option. This will prevent compilation and linking of the Default Chip Driver List (*default_registered_chips*) provided by IDF. Instead, the linker will search for the structure of the same name (*default_registered_chips*) that must be provided by the user.
2. Add a new component in your project, e.g. *custom_chip_driver*.
3. Copy the necessary chip driver files from the *spi_flash* component in IDF. This may include:
   - *spi_flash_chip_drivers.c* (to provide the *default_registered_chips* structure)
   - Any of the *spi_flash_chip_*.c* files that matches your own flash model best
   - *CMakeLists.txt* and *linker.lf* files

Modify the files above properly. Including:

- Change the `default_registered_chips` variable to non-static and remove the #ifdef logic around it.
- Update *linker.lf* file to rename the fragment header and the library name to match the new component.
- If reusing other drivers, some header names need prefixing with `spi_flash/` when included from outside spi_flash component.

Note:

- When writing your own flash chip driver, you can set your flash chip capabilities through *spi_flash_chip_***(vendor)_get_caps* and points the function pointer *get_chip_caps* for protection to the *spi_flash_chip_***_get_caps* function. The steps are as follows.
    1. Please check whether your flash chip have the capabilities listed in *spi_flash_caps_t* by checking the flash datasheet.
    2. Write a function named *spi_flash_chip_***(vendor)_get_caps*. Take the example below as a reference. (if the flash support *suspend* and *read unique id*).
    3. Points the the pointer *get_chip_caps* (in *spi_flash_chip_t*) to the function mentioned above.

```
spi_flash_caps_t spi_flash_chip_***(vendor)_get_caps(esp_flash_t *chip)
{
    spi_flash_caps_t caps_flags = 0;
    // 32-bit-address flash is not supported
    flash-suspend is supported
    caps_flags |= SPI_FLAHS_CHIP_CAP_SUSPEND;
    // flash read unique id.
    caps_flags |= SPI_FLASH_CHIP_CAP_UNIQUE_ID;
    return caps_flags;
}
```

```
const spi_flash_chip_t esp_flash_chip_eon = {
    // Other function pointers
    .get_chip_caps = spi_flash_chip_eon_get_caps,
};
```

- You also can see how to implement this in the example storage/custom_flash_driver.

---

4. Write a new *CMakeLists.txt* file for the *custom_chip_driver* component, including an additional line to add a linker dependency from *spi_flash* to *custom_chip_driver*:

```
idf_component_register(SRCS "spi_flash_chip_drivers.c"
                       "spi_flash_chip_mychip.c"  # modify as needed
                       REQUIRES hal
                       PRIV_REQUIRES spi_flash
                       LDFRAGMENTS linker.lf)
idf_component_add_link_dependency(FROM spi_flash)
```

    - An example of this component CMakeLists.txt can be found in storage/custom_flash_driver/components/custom_chip_driver/CMakeLists.txt
5. The *linker.lf* is used to put every chip driver that you are going to use whilst cache is disabled into internal RAM. See *Linker Script Generation* for more details. Make sure this file covers all the source files that you add.
6. Build your project, and you will see the new flash driver is used.

Example See also storage/custom_flash_driver.

Initializing a Flash Device

To use the `esp_flash_*` APIs, you need to initialise a flash chip on a certain SPI bus, as shown below:

1. Call *`spi_bus_initialize()`* to properly initialize an SPI bus. This function initializes the resources (I/O, DMA, interrupts) shared among devices attached to this bus.

2. Call *spi_bus_add_flash_device()* to attach the flash device to the bus. This function allocates memory and fills the members for the esp_flash_t structure. The CS I/O is also initialized here.
3. Call *esp_flash_init()* to actually communicate with the chip. This will also detect the chip type, and influence the following operations.

Note: Multiple flash chips can be attached to the same bus now.

SPI Flash Access API

This is the set of API functions for working with data in flash:

- *esp_flash_read()* reads data from flash to RAM
- *esp_flash_write()* writes data from RAM to flash
- *esp_flash_erase_region()* erases specific region of flash
- *esp_flash_erase_chip()* erases the whole flash
- esp_flash_get_chip_size() returns flash chip size, in bytes, as configured in menuconfig

Generally, try to avoid using the raw SPI flash functions to the "main" SPI flash chip in favour of *partition-specific functions*.

SPI Flash Size

The SPI flash size is configured by writing a field in the software bootloader image header, flashed at offset 0x1000.

By default, the SPI flash size is detected by esptool.py when this bootloader is written to flash, and the header is updated with the correct size. Alternatively, it is possible to generate a fixed flash size by setting CONFIG_ESPTOOLPY_FLASHSIZE in the project configuration.

If it is necessary to override the configured flash size at runtime, it is possible to set the chip_size member of the g_rom_flashchip structure. This size is used by esp_flash_* functions (in both software & ROM) to check the bounds.

Concurrency Constraints for Flash on SPI1

Concurrency Constraints for flash on SPI1 The SPI0/1 bus is shared between the instruction & data cache (for firmware execution) and the SPI1 peripheral (controlled by the drivers including this SPI Flash driver). Hence, operations to SPI1 will cause significant influence to the whole system. This kind of operations include calling SPI Flash API or other drivers on SPI1 bus, any operations like read/write/erase or other user defined SPI operations, regardless to the main flash or other SPI slave devices.

On ESP32, these caches must be disabled while reading/writing/erasing.

When the caches are disabled Under this condition, all CPUs should always execute code and access data from internal RAM. The APIs documented in this file will disable the caches automatically and transparently.

The way that these APIs disable the caches will suspend all the other tasks. Besides, all non-IRAM-safe interrupts will be disabled. The other core will be polling in a busy loop. These will be restored until the Flash operation completes.

See also *OS Functions* and *SPI Bus Lock*.

There are no such constraints and impacts for flash chips on other SPI buses than SPI0/1.

For differences between internal RAM (e.g. IRAM, DRAM) and flash cache, please refer to the *application memory layout* documentation.

IRAM-Safe Interrupt Handlers For interrupt handlers which need to execute when the cache is disabled (e.g., for low latency operations), set the `ESP_INTR_FLAG_IRAM` flag when the *interrupt handler is registered*.

You must ensure that all data and functions accessed by these interrupt handlers, including the ones that handlers call, are located in IRAM or DRAM. See *How to place code in IRAM*.

If a function or symbol is not correctly put into IRAM/DRAM, and the interrupt handler reads from the flash cache during a flash operation, it will cause a crash due to Illegal Instruction exception (for code which should be in IRAM) or garbage data to be read (for constant data which should be in DRAM).

Note: When working with strings in ISRs, it is not advised to use `printf` and other output functions. For debugging purposes, use `ESP_DRAM_LOGE()` and similar macros when logging from ISRs. Make sure that both `TAG` and format string are placed into `DRAM` in that case.

Non-IRAM-Safe Interrupt Handlers If the `ESP_INTR_FLAG_IRAM` flag is not set when registering, the interrupt handler will not get executed when the caches are disabled. Once the caches are restored, the non-IRAM-safe interrupts will be re-enabled. After this moment, the interrupt handler will run normally again. This means that as long as caches are disabled, users won't see the corresponding hardware event happening.

Attention: The SPI0/1 bus is shared between the instruction & data cache (for firmware execution) and the SPI1 peripheral (controlled by the drivers including this SPI flash driver). Hence, calling SPI Flash API on SPI1 bus (including the main flash) will cause significant influence to the whole system. See *Concurrency Constraints for flash on SPI1* for more details.

Partition Table API

ESP-IDF projects use a partition table to maintain information about various regions of SPI flash memory (bootloader, various application binaries, data, filesystems). More information can be found in *Partition Tables*.

This component provides API functions to enumerate partitions found in the partition table and perform operations on them. These functions are declared in `esp_partition.h`:

- *`esp_partition_find()`* checks a partition table for entries with specific type, returns an opaque iterator.
- *`esp_partition_get()`* returns a structure describing the partition for a given iterator.
- *`esp_partition_next()`* shifts the iterator to the next found partition.
- *`esp_partition_iterator_release()`* releases iterator returned by `esp_partition_find`.
- *`esp_partition_find_first()`* is a convenience function which returns the structure describing the first partition found by `esp_partition_find`.
- *`esp_partition_read()`*, *`esp_partition_write()`*, *`esp_partition_erase_range()`* are equivalent to *`esp_flash_read()`*, *`esp_flash_write()`*, *`esp_flash_erase_region()`*, but operate within partition boundaries.

Note: Application code should mostly use these `esp_partition_*` API functions instead of lower level `esp_flash_*` API functions. Partition table API functions do bounds checking and calculate correct offsets in flash, based on data stored in a partition table.

SPI Flash Encryption

It is possible to encrypt the contents of SPI flash and have it transparently decrypted by hardware.

Refer to the *Flash Encryption documentation* for more details.

Memory Mapping API

ESP32 features memory hardware which allows regions of flash memory to be mapped into instruction and data address spaces. This mapping works only for read operations. It is not possible to modify contents of flash memory by writing to a mapped memory region.

Mapping happens in 64 KB pages. Memory mapping hardware can map flash into the data address space and the instruction address space. See the technical reference manual for more details and limitations about memory mapping hardware.

Note that some pages are used to map the application itself into memory, so the actual number of available pages may be less than the capability of the hardware.

Reading data from flash using a memory mapped region is the only way to decrypt contents of flash when *flash encryption* is enabled. Decryption is performed at the hardware level.

Memory mapping API are declared in `spi_flash_mmap.h` and `esp_partition.h`:

- *`spi_flash_mmap()`* maps a region of physical flash addresses into instruction space or data space of the CPU.
- *`spi_flash_munmap()`* unmaps previously mapped region.
- *`esp_partition_mmap()`* maps part of a partition into the instruction space or data space of the CPU.

Differences between *`spi_flash_mmap()`* and *`esp_partition_mmap()`* are as follows:

- *`spi_flash_mmap()`* must be given a 64 KB aligned physical address.
- *`esp_partition_mmap()`* may be given any arbitrary offset within the partition. It will adjust the returned pointer to mapped memory as necessary.

Note that since memory mapping happens in pages, it may be possible to read data outside of the partition provided to `esp_partition_mmap`, regardless of the partition boundary.

Note: mmap is supported by cache, so it can only be used on main flash.

SPI Flash Implementation

The `esp_flash_t` structure holds chip data as well as three important parts of this API:

1. The host driver, which provides the hardware support to access the chip;
2. The chip driver, which provides compatibility service to different chips;
3. The OS functions, provide support of some OS functions (e.g. lock, delay) in different stages (1st/2nd boot, or the app).

Host driver The host driver relies on an interface (`spi_flash_host_driver_t`) defined in the `spi_flash_types.h` (in the `hal/include/hal` folder). This interface provides some common functions to communicate with the chip.

In other files of the SPI HAL, some of these functions are implemented with existing ESP32 memory-spi functionalities. However, due to the speed limitations of ESP32, the HAL layer cannot provide high-speed implementations to some reading commands (so the support for it was dropped). The files (`memspi_host_driver.h` and `.c`) implement the high-speed version of these commands with the `common_command` function provided in the HAL, and wrap these functions as `spi_flash_host_driver_t` for upper layer to use.

You can also implement your own host driver, even with the GPIO. As long as all the functions in the `spi_flash_host_driver_t` are implemented, the esp_flash API can access the flash regardless of the low-level hardware.

Chip Driver The chip driver, defined in `spi_flash_chip_driver.h`, wraps basic functions provided by the host driver for the API layer to use.

Some operations need some commands to be sent first, or read some status afterwards. Some chips need different commands or values, or need special communication ways.

There is a type of chip called `generic chip` which stands for common chips. Other special chip drivers can be developed on the base of the generic chip.

The chip driver relies on the host driver.

OS Functions Currently the OS function layer provides entries of a lock and delay.

The lock (see *SPI Bus Lock*) is used to resolve the conflicts among the access of devices on the same SPI bus, and the SPI Flash chip access. E.g.

1. On SPI1 bus, the cache (used to fetch the data (code) in the Flash and PSRAM) should be disabled when the flash chip on the SPI0/1 is being accessed.
2. On the other buses, the flash driver needs to disable the ISR registered by SPI Master driver, to avoid conflicts.
3. Some devices of SPI Master driver may require to use the bus monopolized during a period (especially when the device doesn't have a CS wire, or the wire is controlled by software like SDSPI driver).

The delay is used by some long operations which requires the master to wait or polling periodically.

The top API wraps these the chip driver and OS functions into an entire component, and also provides some argument checking.

OS functions can also help to avoid a watchdog timeout when erasing large flash areas. During this time, the CPU is occupied with the flash erasing task. This stops other tasks from being executed. Among these tasks is the idle task to feed the watchdog timer (WDT). If the configuration option *CONFIG_ESP_TASK_WDT_PANIC* is selected and the flash operation time is longer than the watchdog timeout period, the system will reboot.

It's pretty hard to totally eliminate this risk, because the erasing time varies with different flash chips, making it hard to be compatible in flash drivers. Therefore, users need to pay attention to it. Please use the following guidelines:

1. It is recommended to enable the *CONFIG_SPI_FLASH_YIELD_DURING_ERASE* option to allow the scheduler to re-schedule during erasing flash memory. Besides, following parameters can also be used.

   - Increase *CONFIG_SPI_FLASH_ERASE_YIELD_TICKS* or decrease *CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS* in menuconfig.
   - You can also increase *CONFIG_ESP_TASK_WDT_TIMEOUT_S* in menuconfig for a larger watchdog timeout period. However, with larger watchdog timeout period, previously detected timeouts may no longer be detected.

2. Please be aware of the consequences of enabling the *CONFIG_ESP_TASK_WDT_PANIC* option when doing long-running SPI flash operations which will trigger the panic handler when it times out. However, this option can also help dealing with unexpected exceptions in your application. Please decide whether this is needed to be enabled according to actual condition.
3. During your development, please carefully review the actual flash operation according to the specific requirements and time limits on erasing flash memory of your projects. Always allow reasonable redundancy based on your specific product requirements when configuring the flash erasing timeout threshold, thus improving the reliability of your product.

See Also

- *Partition Table documentation*
- *Over The Air Update (OTA) API* provides high-level API for updating app firmware stored in flash.
- *Non-Volatile Storage (NVS) API* provides a structured API for storing small pieces of data in SPI flash.

Implementation Details

In order to perform some flash operations, it is necessary to make sure that both CPUs are not running any code from flash for the duration of the flash operation: - In a single-core setup, the SDK needs to disable interrupts or scheduler

before performing the flash operation. - In a dual-core setup, the SDK needs to make sure that both CPUs are not running any code from flash.

When SPI flash API is called on CPU A (can be PRO or APP), start the `spi_flash_op_block_func` function on CPU B using the `esp_ipc_call` API. This API wakes up a high priority task on CPU B and tells it to execute a given function, in this case, `spi_flash_op_block_func`. This function disables cache on CPU B and signals that the cache is disabled by setting the `s_flash_op_can_start` flag. Then the task on CPU A disables cache as well and proceeds to execute flash operation.

While a flash operation is running, interrupts can still run on CPUs A and B. It is assumed that all interrupt code is placed into RAM. Once the interrupt allocation API is added, a flag should be added to request the interrupt to be disabled for the duration of a flash operations.

Once the flash operation is complete, the function on CPU A sets another flag, `s_flash_op_complete`, to let the task on CPU B know that it can re-enable cache and release the CPU. Then the function on CPU A re-enables the cache on CPU A as well and returns control to the calling code.

Additionally, all API functions are protected with a mutex (`s_flash_op_mutex`).

In a single core environment (*CONFIG_FREERTOS_UNICORE* enabled), you need to disable both caches, so that no inter-CPU communication can take place.

API Reference - SPI Flash

Header File

- components/spi_flash/include/esp_flash_spi_init.h

Functions

esp_err_t **spi_bus_add_flash_device**(*esp_flash_t* **out_chip, const *esp_flash_spi_device_config_t* *config)

Add a SPI Flash device onto the SPI bus.

The bus should be already initialized by `spi_bus_initialization`.

Parameters
- **out_chip** –Pointer to hold the initialized chip.
- **config** –Configuration of the chips to initialize.

Returns
- ESP_ERR_INVALID_ARG: out_chip is NULL, or some field in the config is invalid.
- ESP_ERR_NO_MEM: failed to allocate memory for the chip structures.
- ESP_OK: success.

esp_err_t **spi_bus_remove_flash_device**(*esp_flash_t* *chip)

Remove a SPI Flash device from the SPI bus.

Parameters **chip** –The flash device to remove.

Returns
- ESP_ERR_INVALID_ARG: The chip is invalid.
- ESP_OK: success.

Structures

struct **esp_flash_spi_device_config_t**

Configurations for the SPI Flash to init.

Public Members

spi_host_device_t **host_id**

Bus to use.

int **cs_io_num**

GPIO pin to output the CS signal.

esp_flash_io_mode_t **io_mode**

IO mode to read from the Flash.

enum *esp_flash_speed_s* **speed**

Speed of the Flash clock. Replaced by freq_mhz.

int **input_delay_ns**

Input delay of the data pins, in ns. Set to 0 if unknown.

int **cs_id**

CS line ID, ignored when not `host_id` is not SPI1_HOST, or `CONFIG_SPI_FLASH_SHARE_SPI1_BUS` is enabled. In this case, the CS line used is automatically assigned by the SPI bus lock.

int **freq_mhz**

The frequency of flash chip(MHZ)

Header File

- components/spi_flash/include/esp_flash.h

Functions

esp_err_t **esp_flash_init**(*esp_flash_t* *chip)

Initialise SPI flash chip interface.

This function must be called before any other API functions are called for this chip.

Note: Only the `host` and `read_mode` fields of the chip structure must be initialised before this function is called. Other fields may be auto-detected if left set to zero or NULL.

Note: If the chip->drv pointer is NULL, chip chip_drv will be auto-detected based on its manufacturer & product IDs. See `esp_flash_registered_flash_drivers` pointer for details of this process.

Parameters **chip** –Pointer to SPI flash chip to use. If NULL, esp_flash_default_chip is substituted.

Returns ESP_OK on success, or a flash error code if initialisation fails.

bool **esp_flash_chip_driver_initialized**(const *esp_flash_t* *chip)

Check if appropriate chip driver is set.

Parameters **chip** –Pointer to SPI flash chip to use. If NULL, esp_flash_default_chip is substituted.

Returns true if set, otherwise false.

esp_err_t **esp_flash_read_id**(*esp_flash_t* *chip, uint32_t *out_id)

Read flash ID via the common "RDID" SPI flash command.

ID is a 24-bit value. Lower 16 bits of 'id' are the chip ID, upper 8 bits are the manufacturer ID.

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **out_id** –**[out]** Pointer to receive ID value.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_size**(*esp_flash_t* *chip, uint32_t *out_size)

Detect flash size based on flash ID.

Note: 1. Most flash chips use a common format for flash ID, where the lower 4 bits specify the size as a power of 2. If the manufacturer doesn't follow this convention, the size may be incorrectly detected.

a. The out_size returned only stands for The out_size stands for the size in the binary image header. If you want to get the real size of the chip, please call `esp_flash_get_physical_size` instead.

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **out_size** –**[out]** Detected size in bytes, standing for the size in the binary image header.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_physical_size**(*esp_flash_t* *chip, uint32_t *flash_size)

Detect flash size based on flash ID.

Note: Most flash chips use a common format for flash ID, where the lower 4 bits specify the size as a power of 2. If the manufacturer doesn't follow this convention, the size may be incorrectly detected.

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **flash_size** –**[out]** Detected size in bytes.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_read_unique_chip_id**(*esp_flash_t* *chip, uint64_t *out_id)

Read flash unique ID via the common "RDUID" SPI flash command.

ID is a 64-bit value.

Note: This is an optional feature, which is not supported on all flash chips. READ PROGRAMMING GUIDE FIRST!

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init().
- **out_id** –**[out]** Pointer to receive unique ID value.

Returns

- ESP_OK on success, or a flash error code if operation failed.

- ESP_ERR_NOT_SUPPORTED if the chip doesn't support read id.

esp_err_t **esp_flash_erase_chip**(*esp_flash_t* *chip)

Erase flash chip contents.

Parameters **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()

Returns

- ESP_OK on success,
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- Other flash error code if operation failed.

esp_err_t **esp_flash_erase_region**(*esp_flash_t* *chip, uint32_t start, uint32_t len)

Erase a region of the flash chip.

Sector size is specifyed in chip->drv->sector_size field (typically 4096 bytes.) ESP_ERR_INVALID_ARG will be returned if the start & length are not a multiple of this size.

Erase is performed using block (multi-sector) erases where possible (block size is specified in chip->drv->block_erase_size field, typically 65536 bytes). Remaining sectors are erased using individual sector erase commands.

Parameters

- **chip** –Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. Must have been successfully initialised via esp_flash_init()
- **start** –Address to start erasing flash. Must be sector aligned.
- **len** –Length of region to erase. Must also be sector aligned.

Returns

- ESP_OK on success,
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- Other flash error code if operation failed.

esp_err_t **esp_flash_get_chip_write_protect**(*esp_flash_t* *chip, bool *write_protected)

Read if the entire chip is write protected.

Note: A correct result for this flag depends on the SPI flash chip model and chip_drv in use (via the 'chip->drv' field).

Parameters

- **chip** –Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. Must have been successfully initialised via esp_flash_init()
- **write_protected** –**[out]** Pointer to boolean, set to the value of the write protect flag.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_set_chip_write_protect**(*esp_flash_t* *chip, bool write_protect)

Set write protection for the SPI flash chip.

Some SPI flash chips may require a power cycle before write protect status can be cleared. Otherwise, write protection can be removed via a follow-up call to this function.

Note: Correct behaviour of this function depends on the SPI flash chip model and chip_drv in use (via the 'chip->drv' field).

Parameters

- **chip** –Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. Must have been successfully initialised via esp_flash_init()
- **write_protect** –Boolean value for the write protect flag

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_protectable_regions**(const *esp_flash_t* *chip, const *esp_flash_region_t* **out_regions, uint32_t *out_num_regions)

Read the list of individually protectable regions of this SPI flash chip.

Note: Correct behaviour of this function depends on the SPI flash chip model and chip_drv in use (via the 'chip->drv' field).

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **out_regions** –**[out]** Pointer to receive a pointer to the array of protectable regions of the chip.
- **out_num_regions** –**[out]** Pointer to an integer receiving the count of protectable regions in the array returned in 'regions'.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_protected_region**(*esp_flash_t* *chip, const *esp_flash_region_t* *region, bool *out_protected)

Detect if a region of the SPI flash chip is protected.

Note: It is possible for this result to be false and write operations to still fail, if protection is enabled for the entire chip.

Note: Correct behaviour of this function depends on the SPI flash chip model and chip_drv in use (via the 'chip->drv' field).

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **region** –Pointer to a struct describing a protected region. This must match one of the regions returned from esp_flash_get_protectable_regions(···).
- **out_protected** –**[out]** Pointer to a flag which is set based on the protected status for this region.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_set_protected_region**(*esp_flash_t* *chip, const *esp_flash_region_t* *region, bool protect)

Update the protected status for a region of the SPI flash chip.

Note: It is possible for the region protection flag to be cleared and write operations to still fail, if protection is enabled for the entire chip.

Note: Correct behaviour of this function depends on the SPI flash chip model and chip_drv in use (via the 'chip->drv' field).

Parameters

- **chip** –Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- **region** –Pointer to a struct describing a protected region. This must match one of the regions returned from esp_flash_get_protectable_regions(…).
- **protect** –Write protection flag to set.

Returns ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_read**(*esp_flash_t* *chip, void *buffer, uint32_t address, uint32_t length)

Read data from the SPI flash chip.

There are no alignment constraints on buffer, address or length.

Note: If on-chip flash encryption is used, this function returns raw (ie encrypted) data. Use the flash cache to transparently decrypt data.

Parameters

- **chip** –Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. Must have been successfully initialised via esp_flash_init()
- **buffer** –Pointer to a buffer where the data will be read. To get better performance, this should be in the DRAM and word aligned.
- **address** –Address on flash to read from. Must be less than chip->size field.
- **length** –Length (in bytes) of data to read.

Returns

- ESP_OK: success
- ESP_ERR_NO_MEM: Buffer is in external PSRAM which cannot be concurrently accessed, and a temporary internal buffer could not be allocated.
- or a flash error code if operation failed.

esp_err_t **esp_flash_write**(*esp_flash_t* *chip, const void *buffer, uint32_t address, uint32_t length)

Write data to the SPI flash chip.

There are no alignment constraints on buffer, address or length.

Parameters

- **chip** –Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. Must have been successfully initialised via esp_flash_init()
- **address** –Address on flash to write to. Must be previously erased (SPI NOR flash can only write bits 1->0).
- **buffer** –Pointer to a buffer with the data to write. To get better performance, this should be in the DRAM and word aligned.
- **length** –Length (in bytes) of data to write.

Returns

- ESP_OK on success,
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- Other flash error code if operation failed.

esp_err_t **esp_flash_write_encrypted**(*esp_flash_t* *chip, uint32_t address, const void *buffer, uint32_t length)

Encrypted and write data to the SPI flash chip using on-chip hardware flash encryption.

Note: Both address & length must be 16 byte aligned, as this is the encryption block size

---

Parameters

- **chip** –Pointer to identify flash chip. Must be NULL (the main flash chip). For other chips, encrypted write is not supported.
- **address** –Address on flash to write to. 16 byte aligned. Must be previously erased (SPI NOR flash can only write bits 1->0).
- **buffer** –Pointer to a buffer with the data to write.
- **length** –Length (in bytes) of data to write. 16 byte aligned.

Returns

- ESP_OK: on success
- ESP_ERR_NOT_SUPPORTED: encrypted write not supported for this chip.
- ESP_ERR_INVALID_ARG: Either the address, buffer or length is invalid.

esp_err_t **esp_flash_read_encrypted**(*esp_flash_t* *chip, uint32_t address, void *out_buffer, uint32_t length)

Read and decrypt data from the SPI flash chip using on-chip hardware flash encryption.

Parameters

- **chip** –Pointer to identify flash chip. Must be NULL (the main flash chip). For other chips, encrypted read is not supported.
- **address** –Address on flash to read from.
- **out_buffer** –Pointer to a buffer for the data to read to.
- **length** –Length (in bytes) of data to read.

Returns

- ESP_OK: on success
- ESP_ERR_NOT_SUPPORTED: encrypted read not supported for this chip.

static inline bool **esp_flash_is_quad_mode**(const *esp_flash_t* *chip)

Returns true if chip is configured for Quad I/O or Quad Fast Read.

Parameters **chip** –Pointer to SPI flash chip to use. If NULL, esp_flash_default_chip is substituted.

Returns true if flash works in quad mode, otherwise false

Structures

struct **esp_flash_region_t**

Structure for describing a region of flash.

Public Members

uint32_t **offset**

Start address of this region.

uint32_t **size**

Size of the region.

struct **esp_flash_os_functions_t**

OS-level integration hooks for accessing flash chips inside a running OS.

It's in the public header because some instances should be allocated statically in the startup code. May be updated according to hardware version and new flash chip feature requirements, shouldn't be treated as public API.

For advanced developers, you may replace some of them with your implementations at your own risk.

Public Members

esp_err_t (***start**)(void *arg)

Called before commencing any flash operation. Does not need to be recursive (ie is called at most once for each call to 'end').

esp_err_t (***end**)(void *arg)

Called after completing any flash operation.

esp_err_t (***region_protected**)(void *arg, size_t start_addr, size_t size)

Called before any erase/write operations to check whether the region is limited by the OS

esp_err_t (***delay_us**)(void *arg, uint32_t us)

Delay for at least 'us' microseconds. Called in between 'start' and 'end'.

void *(***get_temp_buffer**)(void *arg, size_t reqest_size, size_t *out_size)

Called for get temp buffer when buffer from application cannot be directly read into/write from.

void (***release_temp_buffer**)(void *arg, void *temp_buf)

Called for release temp buffer.

esp_err_t (***check_yield**)(void *arg, uint32_t chip_status, uint32_t *out_request)

Yield to other tasks. Called during erase operations.

Return ESP_OK means yield needs to be called (got an event to handle), while ESP_ERR_TIMEOUT means skip yield.

esp_err_t (***yield**)(void *arg, uint32_t *out_status)

Yield to other tasks. Called during erase operations.

int64_t (***get_system_time**)(void *arg)

Called for get system time.

void (***set_flash_op_status**)(uint32_t op_status)

Call to set flash operation status

struct **esp_flash_t**

Structure to describe a SPI flash chip connected to the system.

Structure must be initialized before use (passed to esp_flash_init()). It's in the public header because some instances should be allocated statically in the startup code. May be updated according to hardware version and new flash chip feature requirements, shouldn't be treated as public API.

For advanced developers, you may replace some of them with your implementations at your own risk.

Public Members

spi_flash_host_inst_t ***host**

Pointer to hardware-specific "host_driver" structure. Must be initialized before used.

const *spi_flash_chip_t* ***chip_drv**

Pointer to chip-model-specific "adapter" structure. If NULL, will be detected during initialisation.

const *esp_flash_os_functions_t* ***os_func**

Pointer to os-specific hook structure. Call `esp_flash_init_os_functions()` to setup this field, after the host is properly initialized.

void ***os_func_data**

Pointer to argument for os-specific hooks. Left NULL and will be initialized with `os_func`.

esp_flash_io_mode_t **read_mode**

Configured SPI flash read mode. Set before `esp_flash_init` is called.

uint32_t **size**

Size of SPI flash in bytes. If 0, size will be detected during initialisation. Note: this stands for the size in the binary image header. If you want to get the flash physical size, please call `esp_flash_get_physical_size`.

uint32_t **chip_id**

Detected chip id.

uint32_t **busy**

This flag is used to verify chip's status.

uint32_t **hpm_dummy_ena**

This flag is used to verify whether flash works under HPM status.

uint32_t **reserved_flags**

reserved.

Macros

SPI_FLASH_YIELD_REQ_YIELD

SPI_FLASH_YIELD_REQ_SUSPEND

SPI_FLASH_YIELD_STA_RESUME

SPI_FLASH_OS_IS_ERASING_STATUS_FLAG

Type Definitions

typedef struct *spi_flash_chip_t* **spi_flash_chip_t**

typedef struct *esp_flash_t* **esp_flash_t**

Header File

- components/spi_flash/include/spi_flash_mmap.h

Functions

esp_err_t **spi_flash_mmap**(size_t src_addr, size_t size, *spi_flash_mmap_memory_t* memory, const void **out_ptr, *spi_flash_mmap_handle_t* *out_handle)

Map region of flash memory into data or instruction address space.

This function allocates sufficient number of 64kB MMU pages and configures them to map the requested region of flash memory into the address space. It may reuse MMU pages which already provide the required mapping.

As with any allocator, if mmap/munmap are heavily used then the address space may become fragmented. To troubleshoot issues with page allocation, use spi_flash_mmap_dump() function.

Parameters

- **src_addr** –Physical address in flash where requested region starts. This address *must* be aligned to 64kB boundary (SPI_FLASH_MMU_PAGE_SIZE)
- **size** –Size of region to be mapped. This size will be rounded up to a 64kB boundary
- **memory** –Address space where the region should be mapped (data or instruction)
- **out_ptr** –**[out]** Output, pointer to the mapped memory region
- **out_handle** –**[out]** Output, handle which should be used for spi_flash_munmap call

Returns ESP_OK on success, ESP_ERR_NO_MEM if pages can not be allocated

esp_err_t **spi_flash_mmap_pages**(const int *pages, size_t page_count, *spi_flash_mmap_memory_t* memory, const void **out_ptr, *spi_flash_mmap_handle_t* *out_handle)

Map sequences of pages of flash memory into data or instruction address space.

This function allocates sufficient number of 64kB MMU pages and configures them to map the indicated pages of flash memory contiguously into address space. In this respect, it works in a similar way as spi_flash_mmap() but it allows mapping a (maybe non-contiguous) set of pages into a contiguous region of memory.

Parameters

- **pages** –An array of numbers indicating the 64kB pages in flash to be mapped contiguously into memory. These indicate the indexes of the 64kB pages, not the byte-size addresses as used in other functions. Array must be located in internal memory.
- **page_count** –Number of entries in the pages array
- **memory** –Address space where the region should be mapped (instruction or data)
- **out_ptr** –**[out]** Output, pointer to the mapped memory region
- **out_handle** –**[out]** Output, handle which should be used for spi_flash_munmap call

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if pages can not be allocated
- ESP_ERR_INVALID_ARG if pagecount is zero or pages array is not in internal memory

void **spi_flash_munmap**(*spi_flash_mmap_handle_t* handle)

Release region previously obtained using spi_flash_mmap.

Note: Calling this function will not necessarily unmap memory region. Region will only be unmapped when there are no other handles which reference this region. In case of partially overlapping regions it is possible that memory will be unmapped partially.

Parameters **handle** –Handle obtained from spi_flash_mmap

void **spi_flash_mmap_dump**(void)

Display information about mapped regions.

This function lists handles obtained using spi_flash_mmap, along with range of pages allocated to each handle. It also lists all non-zero entries of MMU table and corresponding reference counts.

uint32_t **spi_flash_mmap_get_free_pages**(*spi_flash_mmap_memory_t* memory)

get free pages number which can be mmap

This function will return number of free pages available in mmu table. This could be useful before calling actual spi_flash_mmap (maps flash range to DCache or ICache memory) to check if there is sufficient space available for mapping.

Parameters **memory** –memory type of MMU table free page
Returns number of free pages which can be mmaped

size_t **spi_flash_cache2phys**(const void *cached)

Given a memory address where flash is mapped, return the corresponding physical flash offset.

Cache address does not have have been assigned via spi_flash_mmap(), any address in memory mapped flash space can be looked up.

Parameters **cached** –Pointer to flashed cached memory.
Returns
- SPI_FLASH_CACHE2PHYS_FAIL If cache address is outside flash cache region, or the address is not mapped.
- Otherwise, returns physical offset in flash

const void *__spi_flash_phys2cache__(size_t phys_offs, *spi_flash_mmap_memory_t* memory)

Given a physical offset in flash, return the address where it is mapped in the memory space.

Physical address does not have to have been assigned via spi_flash_mmap(), any address in flash can be looked up.

Note: Only the first matching cache address is returned. If MMU flash cache table is configured so multiple entries point to the same physical address, there may be more than one cache address corresponding to that physical address. It is also possible for a single physical address to be mapped to both the IROM and DROM regions.

Note: This function doesn't impose any alignment constraints, but if memory argument is SPI_FLASH_MMAP_INST and phys_offs is not 4-byte aligned, then reading from the returned pointer will result in a crash.

Parameters
- **phys_offs** –Physical offset in flash memory to look up.
- **memory** –Address space type to look up a flash cache address mapping for (instruction or data)

Returns
- NULL if the physical address is invalid or not mapped to flash cache of the specified memory type.
- Cached memory address (in IROM or DROM space) corresponding to phys_offs.

Macros

ESP_ERR_FLASH_OP_FAIL

This file contains `spi_flash_mmap_xx` APIs, mainly for doing memory mapping to an SPI0-connected external Flash, as well as some helper functions to convert between virtual and physical address

ESP_ERR_FLASH_OP_TIMEOUT

SPI_FLASH_SEC_SIZE

SPI Flash sector size

SPI_FLASH_MMU_PAGE_SIZE

Flash cache MMU mapping page size

SPI_FLASH_CACHE2PHYS_FAIL

Type Definitions

typedef uint32_t **spi_flash_mmap_handle_t**

Opaque handle for memory region obtained from spi_flash_mmap.

Enumerations

enum **spi_flash_mmap_memory_t**

Enumeration which specifies memory space requested in an mmap call.

Values:

enumerator **SPI_FLASH_MMAP_DATA**

map to data memory (Vaddr0), allows byte-aligned access, 4 MB total

enumerator **SPI_FLASH_MMAP_INST**

map to instruction memory (Vaddr1-3), allows only 4-byte-aligned access, 11 MB total

Header File

- components/hal/include/hal/spi_flash_types.h

Structures

struct **spi_flash_trans_t**

Definition of a common transaction. Also holds the return value.

Public Members

uint8_t **reserved**

Reserved, must be 0.

uint8_t **mosi_len**

Output data length, in bytes.

uint8_t **miso_len**

Input data length, in bytes.

uint8_t **address_bitlen**

Length of address in bits, set to 0 if command does not need an address.

uint32_t **address**

Address to perform operation on.

const uint8_t ***mosi_data**

Output data to salve.

uint8_t *__miso_data__

[out] Input data from slave, little endian

uint32_t __flags__

Flags for this transaction. Set to 0 for now.

uint16_t __command__

Command to send.

uint8_t __dummy_bitlen__

Basic dummy bits to use.

uint32_t __io_mode__

Flash working mode when `SPI_FLASH_IGNORE_BASEIO` is specified.

struct __spi_flash_sus_cmd_conf__

Configuration structure for the flash chip suspend feature.

Public Members

uint32_t __sus_mask__

SUS/SUS1/SUS2 bit in flash register.

uint32_t __cmd_rdsr__

Read flash status register(2) command.

uint32_t __sus_cmd__

Flash suspend command.

uint32_t __res_cmd__

Flash resume command.

uint32_t __reserved__

Reserved, set to 0.

struct __spi_flash_encryption_t__

Structure for flash encryption operations.

Public Members

void (*__flash_encryption_enable__)(void)

Enable the flash encryption.

void (*__flash_encryption_disable__)(void)

Disable the flash encryption.

void (*`flash_encryption_data_prepare`)(uint32_t address, const uint32_t *buffer, uint32_t size)

Prepare flash encryption before operation.

Note: address and buffer must be 8-word aligned.

Param address The destination address in flash for the write operation.
Param buffer Data for programming
Param size Size to program.

void (*`flash_encryption_done`)(void)

flash data encryption operation is done.

void (*`flash_encryption_destroy`)(void)

Destroy encrypted result

bool (*`flash_encryption_check`)(uint32_t address, uint32_t length)

Check if is qualified to encrypt the buffer

Param address the address of written flash partition.
Param length Buffer size.

struct `spi_flash_host_inst_t`

SPI Flash Host driver instance

Public Members

const struct *spi_flash_host_driver_s* *`driver`

Pointer to the implementation function table.

struct `spi_flash_host_driver_s`

Host driver configuration and context structure.

Public Members

esp_err_t (*`dev_config`)(*spi_flash_host_inst_t* *host)

Configure the device-related register before transactions. This saves some time to re-configure those registers when we send continuously

esp_err_t (*`common_command`)(*spi_flash_host_inst_t* *host, *spi_flash_trans_t* *t)

Send an user-defined spi transaction to the device.

esp_err_t (*`read_id`)(*spi_flash_host_inst_t* *host, uint32_t *id)

Read flash ID.

void (*`erase_chip`)(*spi_flash_host_inst_t* *host)

Erase whole flash chip.

void (***erase_sector**)(*spi_flash_host_inst_t* *host, uint32_t start_address)

Erase a specific sector by its start address.

void (***erase_block**)(*spi_flash_host_inst_t* *host, uint32_t start_address)

Erase a specific block by its start address.

esp_err_t (***read_status**)(*spi_flash_host_inst_t* *host, uint8_t *out_sr)

Read the status of the flash chip.

esp_err_t (***set_write_protect**)(*spi_flash_host_inst_t* *host, bool wp)

Disable write protection.

void (***program_page**)(*spi_flash_host_inst_t* *host, const void *buffer, uint32_t address, uint32_t length)

Program a page of the flash. Check `max_write_bytes` for the maximum allowed writing length.

bool (***supports_direct_write**)(*spi_flash_host_inst_t* *host, const void *p)

Check whether the SPI host supports direct write.

When cache is disabled, SPI1 doesn't support directly write when buffer isn't internal.

int (***write_data_slicer**)(*spi_flash_host_inst_t* *host, uint32_t address, uint32_t len, uint32_t *align_addr, uint32_t page_size)

Slicer for write data. The `program_page` should be called iteratively with the return value of this function.

Param address Beginning flash address to write
Param len Length request to write
Param align_addr Output of the aligned address to write to
Param page_size Physical page size of the flash chip
Return Length that can be actually written in one `program_page` call

esp_err_t (***read**)(*spi_flash_host_inst_t* *host, void *buffer, uint32_t address, uint32_t read_len)

Read data from the flash. Check `max_read_bytes` for the maximum allowed reading length.

bool (***supports_direct_read**)(*spi_flash_host_inst_t* *host, const void *p)

Check whether the SPI host supports direct read.

When cache is disabled, SPI1 doesn't support directly read when the given buffer isn't internal.

int (***read_data_slicer**)(*spi_flash_host_inst_t* *host, uint32_t address, uint32_t len, uint32_t *align_addr, uint32_t page_size)

Slicer for read data. The `read` should be called iteratively with the return value of this function.

Param address Beginning flash address to read
Param len Length request to read
Param align_addr Output of the aligned address to read
Param page_size Physical page size of the flash chip
Return Length that can be actually read in one `read` call

uint32_t (***host_status**)(*spi_flash_host_inst_t* *host)

Check the host status, 0:busy, 1:idle, 2:suspended.

esp_err_t (***configure_host_io_mode**)(*spi_flash_host_inst_t* *host, uint32_t command, uint32_t addr_bitlen, int dummy_bitlen_base, *esp_flash_io_mode_t* io_mode)

Configure the host to work at different read mode. Responsible to compensate the timing and set IO mode.

void (*`poll_cmd_done`)(*spi_flash_host_inst_t* *host)

Internal use, poll the HW until the last operation is done.

esp_err_t (*`flush_cache`)(*spi_flash_host_inst_t* *host, uint32_t addr, uint32_t size)

For some host (SPI1), they are shared with a cache. When the data is modified, the cache needs to be flushed. Left NULL if not supported.

void (*`check_suspend`)(*spi_flash_host_inst_t* *host)

Suspend check erase/program operation, reserved for ESP32-C3 and ESP32-S3 spi flash ROM IMPL.

void (*`resume`)(*spi_flash_host_inst_t* *host)

Resume flash from suspend manually

void (*`suspend`)(*spi_flash_host_inst_t* *host)

Set flash in suspend status manually

esp_err_t (*`sus_setup`)(*spi_flash_host_inst_t* *host, const *spi_flash_sus_cmd_conf* *sus_conf)

Suspend feature setup for setting cmd and status register mask.

Macros

`SPI_FLASH_TRANS_FLAG_CMD16`

Send command of 16 bits.

`SPI_FLASH_TRANS_FLAG_IGNORE_BASEIO`

Not applying the basic io mode configuration for this transaction.

`SPI_FLASH_TRANS_FLAG_BYTE_SWAP`

Used for DTR mode, to swap the bytes of a pair of rising/falling edge.

`SPI_FLASH_CONFIG_CONF_BITS`

OR the io_mode with this mask, to enable the dummy output feature or replace the first several dummy bits into address to meet the requirements of conf bits. (Used in DIO/QIO/OIO mode)

`SPI_FLASH_OPI_FLAG`

A flag for flash work in opi mode, the io mode below are opi, above are SPI/QSPI mode. DO NOT use this value in any API.

`SPI_FLASH_READ_MODE_MIN`

Slowest io mode supported by ESP32, currently SlowRd.

Type Definitions

typedef enum *esp_flash_speed_s* **`esp_flash_speed_t`**

SPI flash clock speed values, always refer to them by the enum rather than the actual value (more speed may be appended into the list).

A strategy to select the maximum allowed speed is to enumerate from the `ESP_FLSH_SPEED_MAX-1` or highest frequency supported by your flash, and decrease the speed until the probing success.

typedef struct *spi_flash_host_driver_s* **spi_flash_host_driver_t**

Enumerations

enum **esp_flash_speed_s**

SPI flash clock speed values, always refer to them by the enum rather than the actual value (more speed may be appended into the list).

A strategy to select the maximum allowed speed is to enumerate from the `ESP_FLSH_SPEED_MAX-1` or highest frequency supported by your flash, and decrease the speed until the probing success.

Values:

enumerator **ESP_FLASH_5MHZ**

The flash runs under 5MHz.

enumerator **ESP_FLASH_10MHZ**

The flash runs under 10MHz.

enumerator **ESP_FLASH_20MHZ**

The flash runs under 20MHz.

enumerator **ESP_FLASH_26MHZ**

The flash runs under 26MHz.

enumerator **ESP_FLASH_40MHZ**

The flash runs under 40MHz.

enumerator **ESP_FLASH_80MHZ**

The flash runs under 80MHz.

enumerator **ESP_FLASH_120MHZ**

The flash runs under 120MHz, 120MHZ can only be used by main flash after timing tuning in system. Do not use this directely in any API.

enumerator **ESP_FLASH_SPEED_MAX**

The maximum frequency supported by the host is `ESP_FLASH_SPEED_MAX-1`.

enum **esp_flash_io_mode_t**

Mode used for reading from SPI flash.

Values:

enumerator **SPI_FLASH_SLOWRD**

Data read using single I/O, some limits on speed.

enumerator **SPI_FLASH_FASTRD**

Data read using single I/O, no limit on speed.

enumerator **SPI_FLASH_DOUT**

Data read using dual I/O.

enumerator **SPI_FLASH_DIO**

Both address & data transferred using dual I/O.

enumerator **SPI_FLASH_QOUT**

Data read using quad I/O.

enumerator **SPI_FLASH_QIO**

Both address & data transferred using quad I/O.

enumerator **SPI_FLASH_OPI_STR**

Only support on OPI flash, flash read and write under STR mode.

enumerator **SPI_FLASH_OPI_DTR**

Only support on OPI flash, flash read and write under DTR mode.

enumerator **SPI_FLASH_READ_MODE_MAX**

The fastest io mode supported by the host is `ESP_FLASH_READ_MODE_MAX-1`.

Header File

- components/hal/include/hal/esp_flash_err.h

Macros

ESP_ERR_FLASH_NOT_INITIALISED

esp_flash_chip_t structure not correctly initialised by esp_flash_init().

ESP_ERR_FLASH_UNSUPPORTED_HOST

Requested operation isn't supported via this host SPI bus (chip->spi field).

ESP_ERR_FLASH_UNSUPPORTED_CHIP

Requested operation isn't supported by this model of SPI flash chip.

ESP_ERR_FLASH_PROTECTED

Write operation failed due to chip's write protection being enabled.

Enumerations

enum **[anonymous]**

Values:

enumerator **ESP_ERR_FLASH_SIZE_NOT_MATCH**

The chip doesn't have enough space for the current partition table.

enumerator **ESP_ERR_FLASH_NO_RESPONSE**

Chip did not respond to the command, or timed out.

API Reference - Partition Table

Header File

- components/esp_partition/include/esp_partition.h

Functions

esp_partition_iterator_t **esp_partition_find**(*esp_partition_type_t* type, *esp_partition_subtype_t* subtype, const char *label)

Find partition based on one or more parameters.

Parameters

- **type** –Partition type, one of esp_partition_type_t values or an 8-bit unsigned integer. To find all partitions, no matter the type, use ESP_PARTITION_TYPE_ANY, and set subtype argument to ESP_PARTITION_SUBTYPE_ANY.
- **subtype** –Partition subtype, one of esp_partition_subtype_t values or an 8-bit unsigned integer. To find all partitions of given type, use ESP_PARTITION_SUBTYPE_ANY.
- **label** –(optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

Returns iterator which can be used to enumerate all the partitions found, or NULL if no partitions were found. Iterator obtained through this function has to be released using esp_partition_iterator_release when not used any more.

const *esp_partition_t* ***esp_partition_find_first**(*esp_partition_type_t* type, *esp_partition_subtype_t* subtype, const char *label)

Find first partition based on one or more parameters.

Parameters

- **type** –Partition type, one of esp_partition_type_t values or an 8-bit unsigned integer. To find all partitions, no matter the type, use ESP_PARTITION_TYPE_ANY, and set subtype argument to ESP_PARTITION_SUBTYPE_ANY.
- **subtype** –Partition subtype, one of esp_partition_subtype_t values or an 8-bit unsigned integer To find all partitions of given type, use ESP_PARTITION_SUBTYPE_ANY.
- **label** –(optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

Returns pointer to *esp_partition_t* structure, or NULL if no partition is found. This pointer is valid for the lifetime of the application.

const *esp_partition_t* ***esp_partition_get**(*esp_partition_iterator_t* iterator)

Get *esp_partition_t* structure for given partition.

Parameters **iterator** –Iterator obtained using esp_partition_find. Must be non-NULL.
Returns pointer to *esp_partition_t* structure. This pointer is valid for the lifetime of the application.

esp_partition_iterator_t **esp_partition_next**(*esp_partition_iterator_t* iterator)

Move partition iterator to the next partition found.

Any copies of the iterator will be invalid after this call.

Parameters **iterator** –Iterator obtained using esp_partition_find. Must be non-NULL.
Returns NULL if no partition was found, valid esp_partition_iterator_t otherwise.

void **esp_partition_iterator_release**(*esp_partition_iterator_t* iterator)

Release partition iterator.

Parameters **iterator** –Iterator obtained using esp_partition_find. The iterator is allowed to be NULL, so it is not necessary to check its value before calling this function.

const *esp_partition_t* ***esp_partition_verify**(const *esp_partition_t* *partition)

Verify partition data.

Given a pointer to partition data, verify this partition exists in the partition table (all fields match.)

This function is also useful to take partition data which may be in a RAM buffer and convert it to a pointer to the permanent partition data stored in flash.

Pointers returned from this function can be compared directly to the address of any pointer returned from *esp_partition_get()*, as a test for equality.

Parameters **partition** –Pointer to partition data to verify. Must be non-NULL. All fields of this structure must match the partition table entry in flash for this function to return a successful match.

Returns

- If partition not found, returns NULL.
- If found, returns a pointer to the *esp_partition_t* structure in flash. This pointer is always valid for the lifetime of the application.

esp_err_t **esp_partition_read**(const *esp_partition_t* *partition, size_t src_offset, void *dst, size_t size)

Read data from the partition.

Partitions marked with an encryption flag will automatically be be read and decrypted via a cache mapping.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **dst** –Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least ‘size’ bytes long.
- **src_offset** –Address of the data to be read, relative to the beginning of the partition.
- **size** –Size of data to be read, in bytes.

Returns ESP_OK, if data was read successfully; ESP_ERR_INVALID_ARG, if src_offset exceeds partition size; ESP_ERR_INVALID_SIZE, if read would go out of bounds of the partition; or one of error codes from lower-level flash driver.

esp_err_t **esp_partition_write**(const *esp_partition_t* *partition, size_t dst_offset, const void *src, size_t size)

Write data to the partition.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using esp_partition_erase_range function.

Partitions marked with an encryption flag will automatically be written via the esp_flash_write_encrypted() function. If writing to an encrypted partition, all write offsets and lengths must be multiples of 16 bytes. See the esp_flash_write_encrypted() function for more details. Unencrypted partitions do not have this restriction.

Note: Prior to writing to flash memory, make sure it has been erased with esp_partition_erase_range call.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **dst_offset** –Address where the data should be written, relative to the beginning of the partition.
- **src** –Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least ‘size’ bytes long.
- **size** –Size of data to be written, in bytes.

Returns ESP_OK, if data was written successfully; ESP_ERR_INVALID_ARG, if dst_offset exceeds partition size; ESP_ERR_INVALID_SIZE, if write would go out of bounds of the partition; or one of error codes from lower-level flash driver.

esp_err_t **esp_partition_read_raw**(const *esp_partition_t* *partition, size_t src_offset, void *dst, size_t size)

Read data from the partition without any transformation/decryption.

Note: This function is essentially the same as *esp_partition_read()* above. It just never decrypts data but returns it as is.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **dst** –Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- **src_offset** –Address of the data to be read, relative to the beginning of the partition.
- **size** –Size of data to be read, in bytes.

Returns ESP_OK, if data was read successfully; ESP_ERR_INVALID_ARG, if src_offset exceeds partition size; ESP_ERR_INVALID_SIZE, if read would go out of bounds of the partition; or one of error codes from lower-level flash driver.

esp_err_t **esp_partition_write_raw**(const *esp_partition_t* *partition, size_t dst_offset, const void *src, size_t size)

Write data to the partition without any transformation/encryption.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using esp_partition_erase_range function.

Note: This function is essentially the same as *esp_partition_write()* above. It just never encrypts data but writes it as is.

Note: Prior to writing to flash memory, make sure it has been erased with esp_partition_erase_range call.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **dst_offset** –Address where the data should be written, relative to the beginning of the partition.
- **src** –Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- **size** –Size of data to be written, in bytes.

Returns ESP_OK, if data was written successfully; ESP_ERR_INVALID_ARG, if dst_offset exceeds partition size; ESP_ERR_INVALID_SIZE, if write would go out of bounds of the partition; or one of the error codes from lower-level flash driver.

esp_err_t **esp_partition_erase_range**(const *esp_partition_t* *partition, size_t offset, size_t size)

Erase part of the partition.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **offset** –Offset from the beginning of partition where erase operation should start. Must be aligned to partition->erase_size.
- **size** –Size of the range which should be erased, in bytes. Must be divisible by partition->erase_size.

Returns ESP_OK, if the range was erased successfully; ESP_ERR_INVALID_ARG, if iterator or dst are NULL; ESP_ERR_INVALID_SIZE, if erase would go out of bounds of the partition; or one of error codes from lower-level flash driver.

esp_err_t **esp_partition_mmap**(const *esp_partition_t* *partition, size_t offset, size_t size, *esp_partition_mmap_memory_t* memory, const void **out_ptr, *esp_partition_mmap_handle_t* *out_handle)

Configure MMU to map partition into data memory.

Unlike spi_flash_mmap function, which requires a 64kB aligned base address, this function doesn't impose such a requirement. If offset results in a flash address which is not aligned to 64kB boundary, address will be rounded to the lower 64kB boundary, so that mapped region includes requested range. Pointer returned via out_ptr argument will be adjusted to point to the requested offset (not necessarily to the beginning of mmap-ed region).

To release mapped memory, pass handle returned via out_handle argument to esp_partition_munmap function.

Parameters

- **partition** –Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- **offset** –Offset from the beginning of partition where mapping should start.
- **size** –Size of the area to be mapped.
- **memory** –Memory space where the region should be mapped
- **out_ptr** –Output, pointer to the mapped memory region
- **out_handle** –Output, handle which should be used for esp_partition_munmap call

Returns ESP_OK, if successful

void **esp_partition_munmap**(*esp_partition_mmap_handle_t* handle)

Release region previously obtained using esp_partition_mmap.

Note: Calling this function will not necessarily unmap memory region. Region will only be unmapped when there are no other handles which reference this region. In case of partially overlapping regions it is possible that memory will be unmapped partially.

Parameters **handle** –Handle obtained from spi_flash_mmap

esp_err_t **esp_partition_get_sha256**(const *esp_partition_t* *partition, uint8_t *sha_256)

Get SHA-256 digest for required partition.

For apps with SHA-256 appended to the app image, the result is the appended SHA-256 value for the app image content. The hash is verified before returning, if app content is invalid then the function returns ESP_ERR_IMAGE_INVALID. For apps without SHA-256 appended to the image, the result is the SHA-256 of all bytes in the app image. For other partition types, the result is the SHA-256 of the entire partition.

Parameters

- **partition** –**[in]** Pointer to info for partition containing app or data. (fields: address, size and type, are required to be filled).
- **sha_256** –**[out]** Returned SHA-256 digest for a given partition.

Returns

- ESP_OK: In case of successful operation.
- ESP_ERR_INVALID_ARG: The size was 0 or the sha_256 was NULL.
- ESP_ERR_NO_MEM: Cannot allocate memory for sha256 operation.
- ESP_ERR_IMAGE_INVALID: App partition doesn't contain a valid app image.
- ESP_FAIL: An allocation error occurred.

bool **esp_partition_check_identity**(const *esp_partition_t* *partition_1, const *esp_partition_t* *partition_2)

Check for the identity of two partitions by SHA-256 digest.

Parameters

- **partition_1** –**[in]** Pointer to info for partition 1 containing app or data. (fields: address, size and type, are required to be filled).
- **partition_2** –**[in]** Pointer to info for partition 2 containing app or data. (fields: address, size and type, are required to be filled).

Returns

- True: In case of the two firmware is equal.
- False: Otherwise

esp_err_t **esp_partition_register_external**(*esp_flash_t* *flash_chip, size_t offset, size_t size, const char *label, *esp_partition_type_t* type, *esp_partition_subtype_t* subtype, const *esp_partition_t* **out_partition)

Register a partition on an external flash chip.

This API allows designating certain areas of external flash chips (identified by the *esp_flash_t* structure) as partitions. This allows using them with components which access SPI flash through the esp_partition API.

Parameters

- **flash_chip** –Pointer to the structure identifying the flash chip
- **offset** –Address in bytes, where the partition starts
- **size** –Size of the partition in bytes
- **label** –Partition name
- **type** –One of the partition types (ESP_PARTITION_TYPE_*), or an integer. Note that applications can not be booted from external flash chips, so using ESP_PARTITION_TYPE_APP is not supported.
- **subtype** –One of the partition subtypes (ESP_PARTITION_SUBTYPE_*), or an integer.
- **out_partition** –**[out]** Output, if non-NULL, receives the pointer to the resulting *esp_partition_t* structure

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if memory allocation has failed
- ESP_ERR_INVALID_ARG if the new partition overlaps another partition on the same flash chip
- ESP_ERR_INVALID_SIZE if the partition doesn't fit into the flash chip size

esp_err_t **esp_partition_deregister_external**(const *esp_partition_t* *partition)

Deregister the partition previously registered using esp_partition_register_external.

Parameters **partition** –pointer to the partition structure obtained from esp_partition_register_external,

Returns

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition pointer is not found
- ESP_ERR_INVALID_ARG if the partition comes from the partition table
- ESP_ERR_INVALID_ARG if the partition was not registered using esp_partition_register_external function.

Structures

struct **esp_partition_t**

partition information structure

This is not the format in flash, that format is esp_partition_info_t.

However, this is the format used by this API.

Public Members

esp_flash_t ***flash_chip**

SPI flash chip on which the partition resides

esp_partition_type_t **type**

partition type (app/data)

esp_partition_subtype_t **subtype**

partition subtype

uint32_t **address**

starting address of the partition in flash

uint32_t **size**

size of the partition, in bytes

uint32_t **erase_size**

size the erase operation should be aligned to

char **label**[17]

partition label, zero-terminated ASCII string

bool **encrypted**

flag is set to true if partition is encrypted

Macros

ESP_PARTITION_SUBTYPE_OTA(i)

Convenience macro to get esp_partition_subtype_t value for the i-th OTA partition.

Type Definitions

typedef uint32_t **esp_partition_mmap_handle_t**

Opaque handle for memory region obtained from esp_partition_mmap.

typedef struct esp_partition_iterator_opaque_ ***esp_partition_iterator_t**

Opaque partition iterator type.

Enumerations

enum **esp_partition_mmap_memory_t**

Enumeration which specifies memory space requested in an mmap call.

Values:

enumerator **ESP_PARTITION_MMAP_DATA**

map to data memory (Vaddr0), allows byte-aligned access, 4 MB total

enumerator **ESP_PARTITION_MMAP_INST**

map to instruction memory (Vaddr1-3), allows only 4-byte-aligned access, 11 MB total

enum **esp_partition_type_t**

Partition type.

Note: Partition types with integer value 0x00-0x3F are reserved for partition types defined by ESP-IDF. Any other integer value 0x40-0xFE can be used by individual applications, without restriction.

Values:

enumerator **ESP_PARTITION_TYPE_APP**

Application partition type.

enumerator **ESP_PARTITION_TYPE_DATA**

Data partition type.

enumerator **ESP_PARTITION_TYPE_ANY**

Used to search for partitions with any type.

enum **esp_partition_subtype_t**

Partition subtype.

Application-defined partition types (0x40-0xFE) can set any numeric subtype value.

Note: These ESP-IDF-defined partition subtypes apply to partitions of type ESP_PARTITION_TYPE_APP and ESP_PARTITION_TYPE_DATA.

Values:

enumerator **ESP_PARTITION_SUBTYPE_APP_FACTORY**

Factory application partition.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_MIN**

Base for OTA partition subtypes.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_0**

OTA partition 0.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_1**

OTA partition 1.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_2**

OTA partition 2.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_3**

OTA partition 3.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_4**

OTA partition 4.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_5**

OTA partition 5.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_6**
OTA partition 6.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_7**
OTA partition 7.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_8**
OTA partition 8.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_9**
OTA partition 9.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_10**
OTA partition 10.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_11**
OTA partition 11.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_12**
OTA partition 12.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_13**
OTA partition 13.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_14**
OTA partition 14.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_15**
OTA partition 15.

enumerator **ESP_PARTITION_SUBTYPE_APP_OTA_MAX**
Max subtype of OTA partition.

enumerator **ESP_PARTITION_SUBTYPE_APP_TEST**
Test application partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_OTA**
OTA selection partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_PHY**
PHY init data partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_NVS**
NVS partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_COREDUMP**
COREDUMP partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS**

Partition for NVS keys.

enumerator **ESP_PARTITION_SUBTYPE_DATA_EFUSE_EM**

Partition for emulate eFuse bits.

enumerator **ESP_PARTITION_SUBTYPE_DATA_UNDEFINED**

Undefined (or unspecified) data partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD**

ESPHTTPD partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_FAT**

FAT partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_SPIFFS**

SPIFFS partition.

enumerator **ESP_PARTITION_SUBTYPE_DATA_LITTLEFS**

LITTLEFS partition.

enumerator **ESP_PARTITION_SUBTYPE_ANY**

Used to search for partitions with any subtype.

API Reference - Flash Encrypt

Header File

- components/bootloader_support/include/esp_flash_encrypt.h

Functions

bool **esp_flash_encryption_enabled**(void)

Is flash encryption currently enabled in hardware?

Flash encryption is enabled if the FLASH_CRYPT_CNT efuse has an odd number of bits set.

Returns true if flash encryption is enabled.

esp_err_t **esp_flash_encrypt_check_and_update**(void)

bool **esp_flash_encrypt_state**(void)

Returns the Flash Encryption state and prints it.

Returns True - Flash Encryption is enabled False - Flash Encryption is not enabled

bool **esp_flash_encrypt_initialized_once**(void)

Checks if the first initialization was done.

If the first initialization was done then FLASH_CRYPT_CNT != 0

Returns true - the first initialization was done false - the first initialization was NOT done

esp_err_t **esp_flash_encrypt_init**(void)

The first initialization of Flash Encryption key and related eFuses.

Returns ESP_OK if all operations succeeded

esp_err_t **esp_flash_encrypt_contents**(void)

Encrypts flash content.

Returns ESP_OK if all operations succeeded

esp_err_t **esp_flash_encrypt_enable**(void)

Activates Flash encryption on the chip.

It burns FLASH_CRYPT_CNT eFuse based on the CONFIG_SECURE_FLASH_ENCRYPTION_MODE_RELEASE option.

Returns ESP_OK if all operations succeeded

bool **esp_flash_encrypt_is_write_protected**(bool print_error)

Returns True if the write protection of FLASH_CRYPT_CNT is set.

Parameters **print_error** –Print error if it is write protected
Returns true - if FLASH_CRYPT_CNT is write protected

esp_err_t **esp_flash_encrypt_region**(uint32_t src_addr, size_t data_length)

Encrypt-in-place a block of flash sectors.

Note: This function resets RTC_WDT between operations with sectors.

Parameters

- **src_addr** –Source offset in flash. Should be multiple of 4096 bytes.
- **data_length** –Length of data to encrypt in bytes. Will be rounded up to next multiple of 4096 bytes.

Returns ESP_OK if all operations succeeded, ESP_ERR_FLASH_OP_FAIL if SPI flash fails, ESP_ERR_FLASH_OP_TIMEOUT if flash times out.

void **esp_flash_write_protect_crypt_cnt**(void)

Write protect FLASH_CRYPT_CNT.

Intended to be called as a part of boot process if flash encryption is enabled but secure boot is not used. This should protect against serial re-flashing of an unauthorised code in absence of secure boot.

Note: On ESP32 V3 only, write protecting FLASH_CRYPT_CNT will also prevent disabling UART Download Mode. If both are wanted, call esp_efuse_disable_rom_download_mode() before calling this function.

---

esp_flash_enc_mode_t **esp_get_flash_encryption_mode**(void)

Return the flash encryption mode.

The API is called during boot process but can also be called by application to check the current flash encryption mode of ESP32

Returns

void **esp_flash_encryption_init_checks**(void)

Check the flash encryption mode during startup.

Verifies the flash encryption config during startup:

- Correct any insecure flash encryption settings if hardware Secure Boot is enabled.
- Log warnings if the efuse config doesn't match the project config in any way

Note: This function is called automatically during app startup, it doesn't need to be called from the app.

esp_err_t **esp_flash_encryption_enable_secure_features**(void)

Set all secure eFuse features related to flash encryption.

Returns

- ESP_OK - Successfully

bool **esp_flash_encryption_cfg_verify_release_mode**(void)

Returns the verification status for all physical security features of flash encryption in release mode.

If the device has flash encryption feature configured in the release mode, then it is highly recommended to call this API in the application startup code. This API verifies the sanity of the eFuse configuration against the release (production) mode of the flash encryption feature.

Returns

- True - all eFuses are configured correctly
- False - not all eFuses are configured correctly.

void **esp_flash_encryption_set_release_mode**(void)

Switches Flash Encryption from "Development" to "Release".

If already in "Release" mode, the function will do nothing. If flash encryption efuse is not enabled yet then abort. It burns:

- " disable encrypt in dl mode"
- set FLASH_CRYPT_CNT efuse to max

Enumerations

enum **esp_flash_enc_mode_t**

Values:

enumerator **ESP_FLASH_ENC_MODE_DISABLED**

enumerator **ESP_FLASH_ENC_MODE_DEVELOPMENT**

enumerator **ESP_FLASH_ENC_MODE_RELEASE**

2.9.7 SPIFFS Filesystem

Overview

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear levelling, file system consistency checks, and more.

Notes

- Currently, SPIFFS does not support directories, it produces a flat structure. If SPIFFS is mounted under `/spiffs`, then creating a file with the path `/spiffs/tmp/myfile.txt` will create a file called `/tmp/myfile.txt` in SPIFFS, instead of `myfile.txt` in the directory `/spiffs/tmp`.
- It is not a real-time stack. One write operation might take much longer than another.
- For now, it does not detect or handle bad blocks.

- SPIFFS is able to reliably utilize only around 75% of assigned partition space.
- When the filesystem is running out of space, the garbage collector is trying to find free space by scanning the filesystem multiple times, which can take up to several seconds per write function call, depending on required space. This is caused by the SPIFFS design and the issue has been reported multiple times (e.g. here) and in the official SPIFFS github repository. The issue can be partially mitigated by the SPIFFS configuration.
- Deleting a file does not always remove the whole file, which leaves unusable sections throughout the filesystem.
- When the chip experiences a power loss during a file system operation it could result in SPIFFS corruption. However the file system still might be recovered via `esp_spiffs_check` function. More details in the official SPIFFS FAQ.

Tools

spiffsgen.py spiffsgen.py is a write-only Python SPIFFS implementation used to create filesystem images from the contents of a host folder. To use `spiffsgen.py`, open Terminal and run:

```
python spiffsgen.py <image_size> <base_dir> <output_file>
```

The required arguments are as follows:

- **image_size**: size of the partition onto which the created SPIFFS image will be flashed.
- **base_dir**: directory for which the SPIFFS image needs to be created.
- **output_file**: SPIFFS image output file.

There are also other arguments that control image generation. Documentation on these arguments can be found in the tool's help:

```
python spiffsgen.py --help
```

These optional arguments correspond to a possible SPIFFS build configuration. To generate the right image, please make sure that you use the same arguments/configuration as were used to build SPIFFS. As a guide, the help output indicates the SPIFFS build configuration to which the argument corresponds. In cases when these arguments are not specified, the default values shown in the help output will be used.

When the image is created, it can be flashed using `esptool.py` or `parttool.py`.

Aside from invoking the `spiffsgen.py` standalone by manually running it from the command line or a script, it is also possible to invoke `spiffsgen.py` directly from the build system by calling `spiffs_create_partition_image`:

```
spiffs_create_partition_image(<partition> <base_dir> [FLASH_IN_PROJECT] [DEPENDS␣
↪dep dep dep...])
```

This is more convenient as the build configuration is automatically passed to the tool, ensuring that the generated image is valid for that build. An example of this is while the *image_size* is required for the standalone invocation, only the *partition* name is required when using `spiffs_create_partition_image` –the image size is automatically obtained from the project's partition table.

`spiffs_create_partition_image` must be called from one of the component CMakeLists.txt files.

Optionally, users can opt to have the image automatically flashed together with the app binaries, partition tables, etc. on `idf.py flash` by specifying `FLASH_IN_PROJECT`. For example:

```
spiffs_create_partition_image(my_spiffs_partition my_folder FLASH_IN_PROJECT)
```

If FLASH_IN_PROJECT/SPIFFS_IMAGE_FLASH_IN_PROJECT is not specified, the image will still be generated, but you will have to flash it manually using `esptool.py`, `parttool.py`, or a custom build system target.

There are cases where the contents of the base directory itself is generated at build time. Users can use DEPENDS/SPIFFS_IMAGE_DEPENDS to specify targets that should be executed before generating the image:

```
add_custom_target(dep COMMAND ...)

spiffs_create_partition_image(my_spiffs_partition my_folder DEPENDS dep)
```

For an example, see storage/spiffsgen.

mkspiffs Another tool for creating SPIFFS partition images is mkspiffs. Similar to `spiffsgen.py`, it can be used to create an image from a given folder and then flash that image using `esptool.py`

For that, you need to obtain the following parameters:

- **Block Size**: 4096 (standard for SPI Flash)
- **Page Size**: 256 (standard for SPI Flash)
- **Image Size**: Size of the partition in bytes (can be obtained from a partition table)
- **Partition Offset**: Starting address of the partition (can be obtained from a partition table)

To pack a folder into a 1-Megabyte image, run:

```
mkspiffs -c [src_folder] -b 4096 -p 256 -s 0x100000 spiffs.bin
```

To flash the image onto ESP32 at offset 0x110000, run:

```
python esptool.py --chip esp32 --port [port] --baud [baud] write_flash -z 0x110000␣
↪spiffs.bin
```

Notes on which SPIFFS tool to use The two tools presented above offer very similar functionality. However, there are reasons to prefer one over the other, depending on the use case.

Use `spiffsgen.py` in the following cases:

1. If you want to simply generate a SPIFFS image during the build. `spiffsgen.py` makes it very convenient by providing functions/commands from the build system itself.
2. If the host has no C/C++ compiler available, because `spiffsgen.py` does not require compilation.

Use `mkspiffs` in the following cases:

1. If you need to unpack SPIFFS images in addition to image generation. For now, it is not possible with `spiffsgen.py`.
2. If you have an environment where a Python interpreter is not available, but a host compiler is available. Otherwise, a pre-compiled `mkspiffs` binary can do the job. However, there is no build system integration for `mkspiffs` and the user has to do the corresponding work: compiling `mkspiffs` during build (if a pre-compiled binary is not used), creating build rules/targets for the output files, passing proper parameters to the tool, etc.

See also

- *Partition Table documentation*

Application Example

An example of using SPIFFS is provided in the storage/spiffs directory. This example initializes and mounts a SPIFFS partition, then writes and reads data from it using POSIX and C library APIs. See the README.md file in the example directory for more information.

High-level API Reference

Header File

- components/spiffs/include/esp_spiffs.h

Functions

esp_err_t **esp_vfs_spiffs_register** (const *esp_vfs_spiffs_conf_t* *conf)

Register and mount SPIFFS to VFS with given path prefix.

Parameters **conf** –Pointer to *esp_vfs_spiffs_conf_t* configuration structure

Returns

- ESP_OK if success
- ESP_ERR_NO_MEM if objects could not be allocated
- ESP_ERR_INVALID_STATE if already mounted or partition is encrypted
- ESP_ERR_NOT_FOUND if partition for SPIFFS was not found
- ESP_FAIL if mount or format fails

esp_err_t **esp_vfs_spiffs_unregister** (const char *partition_label)

Unregister and unmount SPIFFS from VFS

Parameters **partition_label** –Same label as passed to esp_vfs_spiffs_register.

Returns

- ESP_OK if successful
- ESP_ERR_INVALID_STATE already unregistered

bool **esp_spiffs_mounted** (const char *partition_label)

Check if SPIFFS is mounted

Parameters **partition_label** –Optional, label of the partition to check. If not specified, first partition with subtype=spiffs is used.

Returns

- true if mounted
- false if not mounted

esp_err_t **esp_spiffs_format** (const char *partition_label)

Format the SPIFFS partition

Parameters **partition_label** –Same label as passed to esp_vfs_spiffs_register.

Returns

- ESP_OK if successful
- ESP_FAIL on error

esp_err_t **esp_spiffs_info** (const char *partition_label, size_t *total_bytes, size_t *used_bytes)

Get information for SPIFFS

Parameters

- **partition_label** –Same label as passed to esp_vfs_spiffs_register
- **total_bytes** –**[out]** Size of the file system
- **used_bytes** –**[out]** Current used bytes in the file system

Returns

- ESP_OK if success
- ESP_ERR_INVALID_STATE if not mounted

esp_err_t **esp_spiffs_check** (const char *partition_label)

Check integrity of SPIFFS

Parameters **partition_label** –Same label as passed to esp_vfs_spiffs_register

Returns

- ESP_OK if successful
- ESP_ERR_INVALID_STATE if not mounted
- ESP_FAIL on error

esp_err_t **esp_spiffs_gc**(const char *partition_label, size_t size_to_gc)

Perform garbage collection in SPIFFS partition.

Call this function to run GC and ensure that at least the given amount of space is available in the partition. This function will fail with ESP_ERR_NOT_FINISHED if it is not possible to reclaim the requested space (that is, not enough free or deleted pages in the filesystem). This function will also fail if it fails to reclaim the requested space after CONFIG_SPIFFS_GC_MAX_RUNS number of GC iterations. On one GC iteration, SPIFFS will erase one logical block (4kB). Therefore the value of CONFIG_SPIFFS_GC_MAX_RUNS should be set at least to the maximum expected size_to_gc, divided by 4096. For example, if the application expects to make room for a 1MB file and calls esp_spiffs_gc(label, 1024 * 1024), CONFIG_SPIFFS_GC_MAX_RUNS should be set to at least 256. On the other hand, increasing CONFIG_SPIFFS_GC_MAX_RUNS value increases the maximum amount of time for which any SPIFFS GC or write operation may potentially block.

Parameters

- **partition_label** –Label of the partition to be garbage-collected. The partition must be already mounted.
- **size_to_gc** –The number of bytes that the GC process should attempt to make available.

Returns

- ESP_OK on success
- ESP_ERR_NOT_FINISHED if GC fails to reclaim the size given by size_to_gc
- ESP_ERR_INVALID_STATE if the partition is not mounted
- ESP_FAIL on all other errors

Structures

struct **esp_vfs_spiffs_conf_t**

Configuration structure for esp_vfs_spiffs_register.

Public Members

const char ***base_path**

File path prefix associated with the filesystem.

const char ***partition_label**

Optional, label of SPIFFS partition to use. If set to NULL, first partition with subtype=spiffs will be used.

size_t **max_files**

Maximum files that could be open at the same time.

bool **format_if_mount_failed**

If true, it will format the file system if it fails to mount.

2.9.8 Virtual filesystem component

Overview

Virtual filesystem (VFS) component provides a unified interface for drivers which can perform operations on file-like objects. These can be real filesystems (FAT, SPIFFS, etc.) or device drivers which provide a file-like interface.

This component allows C library functions, such as fopen and fprintf, to work with FS drivers. At a high level, each FS driver is associated with some path prefix. When one of C library functions needs to open a file, the VFS component searches for the FS driver associated with the file path and forwards the call to that driver. VFS also forwards read, write, and other calls for the given file to the same FS driver.

For example, one can register a FAT filesystem driver with the `/fat` prefix and call `fopen("/fat/file.txt", "w")`. The VFS component will then call the function `open` of the FAT driver and pass the argument `/file.txt` to it together with appropriate mode flags. All subsequent calls to C library functions for the returned `FILE*` stream will also be forwarded to the FAT driver.

FS registration

To register an FS driver, an application needs to define an instance of the *esp_vfs_t* structure and populate it with function pointers to FS APIs:

```
esp_vfs_t myfs = {
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
    .open = &myfs_open,
    .fstat = &myfs_fstat,
    .close = &myfs_close,
    .read = &myfs_read,
};

ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Depending on the way how the FS driver declares its API functions, either `read`, `write`, etc., or `read_p`, `write_p`, etc., should be used.

Case 1: API functions are declared without an extra context pointer (the FS driver is a singleton):

```
ssize_t myfs_write(int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
// ... other members initialized

// When registering FS, context pointer (third argument) is NULL:
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Case 2: API functions are declared with an extra context pointer (the FS driver supports multiple instances):

```
ssize_t myfs_write(myfs_t* fs, int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_CONTEXT_PTR,
    .write_p = &myfs_write,
// ... other members initialized

// When registering FS, pass the FS context pointer into the third argument
// (hypothetical myfs_mount function is used for illustrative purposes)
myfs_t* myfs_inst1 = myfs_mount(partition1->offset, partition1->size);
ESP_ERROR_CHECK(esp_vfs_register("/data1", &myfs, myfs_inst1));

// Can register another instance:
myfs_t* myfs_inst2 = myfs_mount(partition2->offset, partition2->size);
ESP_ERROR_CHECK(esp_vfs_register("/data2", &myfs, myfs_inst2));
```

Synchronous input/output multiplexing Synchronous input/output multiplexing by `select()` is supported in the VFS component. The implementation works in the following way.

1. `select()` is called with file descriptors which could belong to various VFS drivers.
2. The file descriptors are divided into groups each belonging to one VFS driver.
3. The file descriptors belonging to non-socket VFS drivers are handed over to the given VFS drivers by `start_select()`, described later on this page. This function represents the driver-specific implementation of `select()` for the given driver. This should be a non-blocking call which means the function should immediately return after setting up the environment for checking events related to the given file descriptors.
4. The file descriptors belonging to the socket VFS driver are handed over to the socket driver by `socket_select()` described later on this page. This is a blocking call which means that it will return only if there is an event related to socket file descriptors or a non-socket driver signals `socket_select()` to exit.
5. Results are collected from each VFS driver and all drivers are stopped by de-initialization of the environment for checking events.
6. The `select()` call ends and returns the appropriate results.

Non-socket VFS drivers If you want to use `select()` with a file descriptor belonging to a non-socket VFS driver, then you need to register the driver with functions `start_select()` and `end_select()` similarly to the following example:

```
// In definition of esp_vfs_t:
    .start_select = &uart_start_select,
    .end_select = &uart_end_select,
// ... other members initialized
```

`start_select()` is called for setting up the environment for detection of read/write/error conditions on file descriptors belonging to the given VFS driver.

`end_select()` is called to stop/deinitialize/free the environment which was setup by `start_select()`.

Note: `end_select()` might be called without a previous `start_select()` call in some rare circumstances. `end_select()` should fail gracefully if this is the case (i.e., should not crash but return an error instead).

Please refer to the reference implementation for the UART peripheral in vfs/vfs_uart.c and most particularly to the functions *`esp_vfs_dev_uart_register()`*, `uart_start_select()`, and `uart_end_select()` for more information.

Please check the following examples that demonstrate the use of `select()` with VFS file descriptors:

- peripherals/uart/uart_select
- system/select

Socket VFS drivers A socket VFS driver is using its own internal implementation of `select()` and non-socket VFS drivers notify it upon read/write/error conditions.

A socket VFS driver needs to be registered with the following functions defined:

```
// In definition of esp_vfs_t:
    .socket_select = &lwip_select,
    .get_socket_select_semaphore = &lwip_get_socket_select_semaphore,
    .stop_socket_select = &lwip_stop_socket_select,
    .stop_socket_select_isr = &lwip_stop_socket_select_isr,
// ... other members initialized
```

`socket_select()` is the internal implementation of `select()` for the socket driver. It works only with file descriptors belonging to the socket VFS.

`get_socket_select_semaphore()` returns the signalization object (semaphore) which will be used in non-socket drivers to stop the waiting in `socket_select()`.

`stop_socket_select()` call is used to stop the waiting in `socket_select()` by passing the object returned by `get_socket_select_semaphore()`.

`stop_socket_select_isr()` has the same functionality as `stop_socket_select()` but it can be used from ISR.

Please see lwip/port/esp32/vfs_lwip.c for a reference socket driver implementation using LWIP.

Note: If you use `select()` for socket file descriptors only then you can disable the *CONFIG_VFS_SUPPORT_SELECT* option to reduce the code size and improve performance. You should not change the socket driver during an active `select()` call or you might experience some undefined behavior.

Paths

Each registered FS has a path prefix associated with it. This prefix can be considered as a "mount point" of this partition.

In case when mount points are nested, the mount point with the longest matching path prefix is used when opening the file. For instance, suppose that the following filesystems are registered in VFS:

- FS 1 on /data
- FS 2 on /data/static

Then:

- FS 1 will be used when opening a file called `/data/log.txt`
- FS 2 will be used when opening a file called `/data/static/index.html`
- Even if `/index.html"` does not exist in FS 2, FS 1 will *not* be searched for `/static/index.html`.

As a general rule, mount point names must start with the path separator (`/`) and must contain at least one character after path separator. However, an empty mount point name is also supported and might be used in cases when an application needs to provide a "fallback" filesystem or to override VFS functionality altogether. Such filesystem will be used if no prefix matches the path given.

VFS does not handle dots (`.`) in path names in any special way. VFS does not treat `..` as a reference to the parent directory. In the above example, using a path `/data/static/../log.txt` will not result in a call to FS 1 to open `/log.txt`. Specific FS drivers (such as FATFS) might handle dots in file names differently.

When opening files, the FS driver receives only relative paths to files. For example:

1. The `myfs` driver is registered with `/data` as a path prefix.
2. The application calls `fopen("/data/config.json", ...)`.
3. The VFS component calls `myfs_open("/config.json", ...)`.
4. The `myfs` driver opens the `/config.json` file.

VFS does not impose any limit on total file path length, but it does limit the FS path prefix to `ESP_VFS_PATH_MAX` characters. Individual FS drivers may have their own filename length limitations.

File descriptors

File descriptors are small positive integers from `0` to `FD_SETSIZE - 1`, where `FD_SETSIZE` is defined in newlib's `sys/types.h`. The largest file descriptors (configured by `CONFIG_LWIP_MAX_SOCKETS`) are reserved for sockets. The VFS component contains a lookup-table called `s_fd_table` for mapping global file descriptors to VFS driver indexes registered in the `s_vfs` array.

Standard IO streams (stdin, stdout, stderr)

If the menuconfig option `UART for console output` is not set to `None`, then `stdin`, `stdout`, and `stderr` are configured to read from, and write to, a UART. It is possible to use UART0 or UART1 for standard IO. By default, UART0 is used with 115200 baud rate; TX pin is GPIO1; RX pin is GPIO3. These parameters can be changed in menuconfig.

Writing to `stdout` or `stderr` will send characters to the UART transmit FIFO. Reading from `stdin` will retrieve characters from the UART receive FIFO.

By default, VFS uses simple functions for reading from and writing to UART. Writes busy-wait until all data is put into UART FIFO, and reads are non-blocking, returning only the data present in the FIFO. Due to this non-blocking read behavior, higher level C library calls, such as `fscanf("%d\n", &var);`, might not have desired results.

Applications which use the UART driver can instruct VFS to use the driver's interrupt driven, blocking read and write functions instead. This can be done using a call to the `esp_vfs_dev_uart_use_driver` function. It is also possible to revert to the basic non-blocking functions using a call to `esp_vfs_dev_uart_use_nonblocking`.

VFS also provides an optional newline conversion feature for input and output. Internally, most applications send and receive lines terminated by the LF (`'\n'`) character. Different terminal programs may require different line termination, such as CR or CRLF. Applications can configure this separately for input and output either via menuconfig, or by calls to the functions `esp_vfs_dev_uart_port_set_rx_line_endings` and `esp_vfs_dev_uart_port_set_tx_line_endings`.

Standard streams and FreeRTOS tasks `FILE` objects for `stdin`, `stdout`, and `stderr` are shared between all FreeRTOS tasks, but the pointers to these objects are stored in per-task `struct _reent`.

The following code is transferred to `fprintf(__getreent()->_stderr, "42\n");` by the preprocessor:

```
fprintf(stderr, "42\n");
```

The `__getreent()` function returns a per-task pointer to `struct _reent` in newlib libc. This structure is allocated on the TCB of each task. When a task is initialized, `_stdin`, `_stdout`, and `_stderr` members of `struct _reent` are set to the values of `_stdin`, `_stdout`, and `_stderr` of `_GLOBAL_REENT` (i.e., the structure which is used before FreeRTOS is started).

Such a design has the following consequences:

- It is possible to set `stdin`, `stdout`, and `stderr` for any given task without affecting other tasks, e.g., by doing `stdin = fopen("/dev/uart/1", "r")`.
- Closing default `stdin`, `stdout`, or `stderr` using `fclose` will close the `FILE` stream object, which will affect all other tasks.
- To change the default `stdin`, `stdout`, `stderr` streams for new tasks, modify `_GLOBAL_REENT->_stdin` (`_stdout`, `_stderr`) before creating the task.

Event fds

`eventfd()` call is a powerful tool to notify a `select()` based loop of custom events. The `eventfd()` implementation in ESP-IDF is generally the same as described in man(2) eventfd except for:

- `esp_vfs_eventfd_register()` has to be called before calling `eventfd()`
- Options `EFD_CLOEXEC`, `EFD_NONBLOCK` and `EFD_SEMAPHORE` are not supported in flags.
- Option `EFD_SUPPORT_ISR` has been added in flags. This flag is required to read and write the eventfd in an interrupt handler.

Note that creating an eventfd with `EFD_SUPPORT_ISR` will cause interrupts to be temporarily disabled when reading, writing the file and during the beginning and the ending of the `select()` when this file is set.

API Reference

Header File

- components/vfs/include/esp_vfs.h

Functions

ssize_t **esp_vfs_write**(struct _reent *r, int fd, const void *data, size_t size)

These functions are to be used in newlib syscall table. They will be called by newlib when it needs to use any of the syscalls.

off_t **esp_vfs_lseek**(struct _reent *r, int fd, off_t size, int mode)

ssize_t **esp_vfs_read**(struct _reent *r, int fd, void *dst, size_t size)

int **esp_vfs_open**(struct _reent *r, const char *path, int flags, int mode)

int **esp_vfs_close**(struct _reent *r, int fd)

int **esp_vfs_fstat**(struct _reent *r, int fd, struct stat *st)

int **esp_vfs_stat**(struct _reent *r, const char *path, struct stat *st)

int **esp_vfs_link**(struct _reent *r, const char *n1, const char *n2)

int **esp_vfs_unlink**(struct _reent *r, const char *path)

int **esp_vfs_rename**(struct _reent *r, const char *src, const char *dst)

int **esp_vfs_utime**(const char *path, const struct utimbuf *times)

esp_err_t **esp_vfs_register**(const char *base_path, const *esp_vfs_t* *vfs, void *ctx)

Register a virtual filesystem for given path prefix.

Parameters

- **base_path** –file path prefix associated with the filesystem. Must be a zero-terminated C string, may be empty. If not empty, must be up to ESP_VFS_PATH_MAX characters long, and at least 2 characters long. Name must start with a "/" and must not end with "/". For example, "/data" or "/dev/spi" are valid. These VFSes would then be called to handle file paths such as "/data/myfile.txt" or "/dev/spi/0". In the special case of an empty base_path, a "fallback" VFS is registered. Such VFS will handle paths which are not matched by any other registered VFS.
- **vfs** –Pointer to *esp_vfs_t*, a structure which maps syscalls to the filesystem driver functions. VFS component doesn't assume ownership of this pointer.
- **ctx** –If vfs->flags has ESP_VFS_FLAG_CONTEXT_PTR set, a pointer which should be passed to VFS functions. Otherwise, NULL.

Returns ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered.

esp_err_t **esp_vfs_register_fd_range**(const *esp_vfs_t* *vfs, void *ctx, int min_fd, int max_fd)

Special case function for registering a VFS that uses a method other than open() to open new file descriptors from the interval <min_fd; max_fd).

This is a special-purpose function intended for registering LWIP sockets to VFS.

Parameters

- **vfs** –Pointer to *esp_vfs_t*. Meaning is the same as for esp_vfs_register().
- **ctx** –Pointer to context structure. Meaning is the same as for esp_vfs_register().
- **min_fd** –The smallest file descriptor this VFS will use.
- **max_fd** –Upper boundary for file descriptors this VFS will use (the biggest file descriptor plus one).

Returns ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered, ESP_ERR_INVALID_ARG if the file descriptor boundaries are incorrect.

esp_err_t **esp_vfs_register_with_id**(const *esp_vfs_t* *vfs, void *ctx, *esp_vfs_id_t* *vfs_id)

Special case function for registering a VFS that uses a method other than open() to open new file descriptors. In comparison with esp_vfs_register_fd_range, this function doesn't pre-registers an interval of file descriptors. File descriptors can be registered later, by using esp_vfs_register_fd.

Parameters

- **vfs** –Pointer to *esp_vfs_t*. Meaning is the same as for esp_vfs_register().

- **ctx** –Pointer to context structure. Meaning is the same as for esp_vfs_register().
- **vfs_id** –Here will be written the VFS ID which can be passed to esp_vfs_register_fd for registering file descriptors.

Returns ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered, ESP_ERR_INVALID_ARG if the file descriptor boundaries are incorrect.

esp_err_t **esp_vfs_unregister**(const char *base_path)

Unregister a virtual filesystem for given path prefix

Parameters **base_path** –file prefix previously used in esp_vfs_register call

Returns ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for given prefix hasn't been registered

esp_err_t **esp_vfs_unregister_with_id**(*esp_vfs_id_t* vfs_id)

Unregister a virtual filesystem with the given index

Parameters **vfs_id** –The VFS ID returned by esp_vfs_register_with_id

Returns ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for the given index hasn't been registered

esp_err_t **esp_vfs_register_fd**(*esp_vfs_id_t* vfs_id, int *fd)

Special function for registering another file descriptor for a VFS registered by esp_vfs_register_with_id.

Parameters

- **vfs_id** –VFS identificator returned by esp_vfs_register_with_id.
- **fd** –The registered file descriptor will be written to this address.

Returns ESP_OK if the registration is successful, ESP_ERR_NO_MEM if too many file descriptors are registered, ESP_ERR_INVALID_ARG if the arguments are incorrect.

esp_err_t **esp_vfs_register_fd_with_local_fd**(*esp_vfs_id_t* vfs_id, int local_fd, bool permanent, int *fd)

Special function for registering another file descriptor with given local_fd for a VFS registered by esp_vfs_register_with_id.

Parameters

- **vfs_id** –VFS identificator returned by esp_vfs_register_with_id.
- **local_fd** –The fd in the local vfs. Passing -1 will set the local fd as the (*fd) value.
- **permanent** –Whether the fd should be treated as permannet (not removed after close())
- **fd** –The registered file descriptor will be written to this address.

Returns ESP_OK if the registration is successful, ESP_ERR_NO_MEM if too many file descriptors are registered, ESP_ERR_INVALID_ARG if the arguments are incorrect.

esp_err_t **esp_vfs_unregister_fd**(*esp_vfs_id_t* vfs_id, int fd)

Special function for unregistering a file descriptor belonging to a VFS registered by esp_vfs_register_with_id.

Parameters

- **vfs_id** –VFS identificator returned by esp_vfs_register_with_id.
- **fd** –File descriptor which should be unregistered.

Returns ESP_OK if the registration is successful, ESP_ERR_INVALID_ARG if the arguments are incorrect.

int **esp_vfs_select**(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)

Synchronous I/O multiplexing which implements the functionality of POSIX select() for VFS.

Parameters

- **nfds** –Specifies the range of descriptors which should be checked. The first nfds descriptors will be checked in each set.
- **readfds** –If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to read, and on output indicates which descriptors are ready to read.
- **writefds** –If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to write, and on output indicates which descriptors are ready to write.

- **errorfds** –If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for error conditions, and on output indicates which descriptors have error conditions.
- **timeout** –If not NULL, then points to timeval structure which specifies the time period after which the functions should time-out and return. If it is NULL, then the function will not time-out. Note that the timeout period is rounded up to the system tick and incremented by one.

Returns The number of descriptors set in the descriptor sets, or -1 when an error (specified by errno) have occurred.

void **esp_vfs_select_triggered**(*esp_vfs_select_sem_t* sem)

Notification from a VFS driver about a read/write/error condition.

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to start_select.

Parameters **sem** –semaphore structure which was passed to the driver by the start_select call

void **esp_vfs_select_triggered_isr**(*esp_vfs_select_sem_t* sem, BaseType_t *woken)

Notification from a VFS driver about a read/write/error condition (ISR version)

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to start_select.

Parameters

- **sem** –semaphore structure which was passed to the driver by the start_select call
- **woken** –is set to pdTRUE if the function wakes up a task with higher priority

ssize_t **esp_vfs_pread**(int fd, void *dst, size_t size, off_t offset)

Implements the VFS layer of POSIX pread()

Parameters

- **fd** –File descriptor used for read
- **dst** –Pointer to the buffer where the output will be written
- **size** –Number of bytes to be read
- **offset** –Starting offset of the read

Returns A positive return value indicates the number of bytes read. -1 is return on failure and errno is set accordingly.

ssize_t **esp_vfs_pwrite**(int fd, const void *src, size_t size, off_t offset)

Implements the VFS layer of POSIX pwrite()

Parameters

- **fd** –File descriptor used for write
- **src** –Pointer to the buffer from where the output will be read
- **size** –Number of bytes to write
- **offset** –Starting offset of the write

Returns A positive return value indicates the number of bytes written. -1 is return on failure and errno is set accordingly.

Structures

struct **esp_vfs_select_sem_t**

VFS semaphore type for select()

Public Members

bool **is_sem_local**

type of “sem” is SemaphoreHandle_t when true, defined by socket driver otherwise

void ***sem**

semaphore instance

struct **esp_vfs_t**

VFS definition structure.

This structure should be filled with pointers to corresponding FS driver functions.

VFS component will translate all FDs so that the filesystem implementation sees them starting at zero. The caller sees a global FD which is prefixed with an pre-filesystem-implementation.

Some FS implementations expect some state (e.g. pointer to some structure) to be passed in as a first argument. For these implementations, populate the members of this structure which have _p suffix, set flags member to ESP_VFS_FLAG_CONTEXT_PTR and provide the context pointer to esp_vfs_register function. If the implementation doesn't use this extra argument, populate the members without _p suffix and set flags member to ESP_VFS_FLAG_DEFAULT.

If the FS driver doesn't provide some of the functions, set corresponding members to NULL.

Public Members

int **flags**

ESP_VFS_FLAG_CONTEXT_PTR or ESP_VFS_FLAG_DEFAULT

ssize_t (***write_p**)(void *p, int fd, const void *data, size_t size)

Write with context pointer

ssize_t (***write**)(int fd, const void *data, size_t size)

Write without context pointer

off_t (***lseek_p**)(void *p, int fd, off_t size, int mode)

Seek with context pointer

off_t (***lseek**)(int fd, off_t size, int mode)

Seek without context pointer

ssize_t (***read_p**)(void *ctx, int fd, void *dst, size_t size)

Read with context pointer

ssize_t (***read**)(int fd, void *dst, size_t size)

Read without context pointer

ssize_t (***pread_p**)(void *ctx, int fd, void *dst, size_t size, off_t offset)

pread with context pointer

ssize_t (***pread**)(int fd, void *dst, size_t size, off_t offset)

pread without context pointer

ssize_t (***pwrite_p**)(void *ctx, int fd, const void *src, size_t size, off_t offset)

pwrite with context pointer

ssize_t (***pwrite**)(int fd, const void *src, size_t size, off_t offset)
pwrite without context pointer

int (***open_p**)(void *ctx, const char *path, int flags, int mode)
open with context pointer

int (***open**)(const char *path, int flags, int mode)
open without context pointer

int (***close_p**)(void *ctx, int fd)
close with context pointer

int (***close**)(int fd)
close without context pointer

int (***fstat_p**)(void *ctx, int fd, struct *stat* *st)
fstat with context pointer

int (***fstat**)(int fd, struct *stat* *st)
fstat without context pointer

int (***stat_p**)(void *ctx, const char *path, struct *stat* *st)
stat with context pointer

int (***stat**)(const char *path, struct *stat* *st)
stat without context pointer

int (***link_p**)(void *ctx, const char *n1, const char *n2)
link with context pointer

int (***link**)(const char *n1, const char *n2)
link without context pointer

int (***unlink_p**)(void *ctx, const char *path)
unlink with context pointer

int (***unlink**)(const char *path)
unlink without context pointer

int (***rename_p**)(void *ctx, const char *src, const char *dst)
rename with context pointer

int (***rename**)(const char *src, const char *dst)
rename without context pointer

DIR *(***opendir_p**)(void *ctx, const char *name)
opendir with context pointer

DIR *(***opendir**)(const char *name)

opendir without context pointer

struct dirent *(***readdir_p**)(void *ctx, DIR *pdir)

readdir with context pointer

struct dirent *(***readdir**)(DIR *pdir)

readdir without context pointer

int (***readdir_r_p**)(void *ctx, DIR *pdir, struct dirent *entry, struct dirent **out_dirent)

readdir_r with context pointer

int (***readdir_r**)(DIR *pdir, struct dirent *entry, struct dirent **out_dirent)

readdir_r without context pointer

long (***telldir_p**)(void *ctx, DIR *pdir)

telldir with context pointer

long (***telldir**)(DIR *pdir)

telldir without context pointer

void (***seekdir_p**)(void *ctx, DIR *pdir, long offset)

seekdir with context pointer

void (***seekdir**)(DIR *pdir, long offset)

seekdir without context pointer

int (***closedir_p**)(void *ctx, DIR *pdir)

closedir with context pointer

int (***closedir**)(DIR *pdir)

closedir without context pointer

int (***mkdir_p**)(void *ctx, const char *name, mode_t mode)

mkdir with context pointer

int (***mkdir**)(const char *name, mode_t mode)

mkdir without context pointer

int (***rmdir_p**)(void *ctx, const char *name)

rmdir with context pointer

int (***rmdir**)(const char *name)

rmdir without context pointer

int (***fcntl_p**)(void *ctx, int fd, int cmd, int arg)

fcntl with context pointer

int (***fcntl**)(int fd, int cmd, int arg)
fcntl without context pointer

int (***ioctl_p**)(void *ctx, int fd, int cmd, va_list args)
ioctl with context pointer

int (***ioctl**)(int fd, int cmd, va_list args)
ioctl without context pointer

int (***fsync_p**)(void *ctx, int fd)
fsync with context pointer

int (***fsync**)(int fd)
fsync without context pointer

int (***access_p**)(void *ctx, const char *path, int amode)
access with context pointer

int (***access**)(const char *path, int amode)
access without context pointer

int (***truncate_p**)(void *ctx, const char *path, off_t length)
truncate with context pointer

int (***truncate**)(const char *path, off_t length)
truncate without context pointer

int (***ftruncate_p**)(void *ctx, int fd, off_t length)
ftruncate with context pointer

int (***ftruncate**)(int fd, off_t length)
ftruncate without context pointer

int (***utime_p**)(void *ctx, const char *path, const struct utimbuf *times)
utime with context pointer

int (***utime**)(const char *path, const struct utimbuf *times)
utime without context pointer

int (***tcsetattr_p**)(void *ctx, int fd, int optional_actions, const struct termios *p)
tcsetattr with context pointer

int (***tcsetattr**)(int fd, int optional_actions, const struct termios *p)
tcsetattr without context pointer

int (***tcgetattr_p**)(void *ctx, int fd, struct termios *p)
tcgetattr with context pointer

int (*`tcgetattr`)(int fd, struct termios *p)

tcgetattr without context pointer

int (*`tcdrain_p`)(void *ctx, int fd)

tcdrain with context pointer

int (*`tcdrain`)(int fd)

tcdrain without context pointer

int (*`tcflush_p`)(void *ctx, int fd, int select)

tcflush with context pointer

int (*`tcflush`)(int fd, int select)

tcflush without context pointer

int (*`tcflow_p`)(void *ctx, int fd, int action)

tcflow with context pointer

int (*`tcflow`)(int fd, int action)

tcflow without context pointer

pid_t (*`tcgetsid_p`)(void *ctx, int fd)

tcgetsid with context pointer

pid_t (*`tcgetsid`)(int fd)

tcgetsid without context pointer

int (*`tcsendbreak_p`)(void *ctx, int fd, int duration)

tcsendbreak with context pointer

int (*`tcsendbreak`)(int fd, int duration)

tcsendbreak without context pointer

esp_err_t (*`start_select`)(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, *esp_vfs_select_sem_t* sem, void **end_select_args)

start_select is called for setting up synchronous I/O multiplexing of the desired file descriptors in the given VFS

int (*`socket_select`)(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)

socket select function for socket FDs with the functionality of POSIX select(); this should be set only for the socket VFS

void (*`stop_socket_select`)(void *sem)

called by VFS to interrupt the socket_select call when select is activated from a non-socket VFS driver; set only for the socket driver

void (*`stop_socket_select_isr`)(void *sem, BaseType_t *woken)

stop_socket_select which can be called from ISR; set only for the socket driver

void *(***get_socket_select_semaphore**)(void)

end_select is called to stop the I/O multiplexing and deinitialize the environment created by start_select for the given VFS

esp_err_t (***end_select**)(void *end_select_args)

get_socket_select_semaphore returns semaphore allocated in the socket driver; set only for the socket driver

Macros

MAX_FDS

Maximum number of (global) file descriptors.

ESP_VFS_PATH_MAX

Maximum length of path prefix (not including zero terminator)

ESP_VFS_FLAG_DEFAULT

Default value of flags member in *esp_vfs_t* structure.

ESP_VFS_FLAG_CONTEXT_PTR

Flag which indicates that FS needs extra context pointer in syscalls.

Type Definitions

typedef int **esp_vfs_id_t**

Header File

- components/vfs/include/esp_vfs_dev.h

Functions

void **esp_vfs_dev_uart_register**(void)

add /dev/uart virtual filesystem driver

This function is called from startup code to enable serial output

void **esp_vfs_dev_uart_set_rx_line_endings**(esp_line_endings_t mode)

Set the line endings expected to be received on UART.

This specifies the conversion between line endings received on UART and newlines (' ' , LF) passed into stdin:

- ESP_LINE_ENDINGS_CRLF: convert CRLF to LF
- ESP_LINE_ENDINGS_CR: convert CR to LF
- ESP_LINE_ENDINGS_LF: no modification

Note: this function is not thread safe w.r.t. reading from UART

Parameters **mode** – line endings expected on UART

void **esp_vfs_dev_uart_set_tx_line_endings**(esp_line_endings_t mode)

Set the line endings to sent to UART.

This specifies the conversion between newlines (’

’ , LF) on stdout and line endings sent over UART:

- ESP_LINE_ENDINGS_CRLF: convert LF to CRLF
- ESP_LINE_ENDINGS_CR: convert LF to CR
- ESP_LINE_ENDINGS_LF: no modification

Note: this function is not thread safe w.r.t. writing to UART

Parameters **mode** –line endings to send to UART

int **esp_vfs_dev_uart_port_set_rx_line_endings**(int uart_num, esp_line_endings_t mode)

Set the line endings expected to be received on specified UART.

This specifies the conversion between line endings received on UART and newlines (’

’ , LF) passed into stdin:

- ESP_LINE_ENDINGS_CRLF: convert CRLF to LF
- ESP_LINE_ENDINGS_CR: convert CR to LF
- ESP_LINE_ENDINGS_LF: no modification

Note: this function is not thread safe w.r.t. reading from UART

Parameters

- **uart_num** –the UART number
- **mode** –line endings to send to UART

Returns 0 if successed, or -1 when an error (specified by errno) have occurred.

int **esp_vfs_dev_uart_port_set_tx_line_endings**(int uart_num, esp_line_endings_t mode)

Set the line endings to sent to specified UART.

This specifies the conversion between newlines (’

’ , LF) on stdout and line endings sent over UART:

- ESP_LINE_ENDINGS_CRLF: convert LF to CRLF
- ESP_LINE_ENDINGS_CR: convert LF to CR
- ESP_LINE_ENDINGS_LF: no modification

Note: this function is not thread safe w.r.t. writing to UART

Parameters

- **uart_num** –the UART number

- **mode** –line endings to send to UART

Returns 0 if successed, or -1 when an error (specified by errno) have occurred.

void **esp_vfs_dev_uart_use_nonblocking**(int uart_num)

set VFS to use simple functions for reading and writing UART Read is non-blocking, write is busy waiting until TX FIFO has enough space. These functions are used by default.

Parameters **uart_num** –UART peripheral number

void **esp_vfs_dev_uart_use_driver**(int uart_num)

set VFS to use UART driver for reading and writing

Note: application must configure UART driver before calling these functions With these functions, read and write are blocking and interrupt-driven.

Parameters **uart_num** –UART peripheral number

void **esp_vfs_usb_serial_jtag_use_driver**(void)

set VFS to use USB-SERIAL-JTAG driver for reading and writing

---

Note: application must configure USB-SERIAL-JTAG driver before calling these functions With these functions, read and write are blocking and interrupt-driven.

---

void **esp_vfs_usb_serial_jtag_use_nonblocking**(void)

set VFS to use simple functions for reading and writing UART Read is non-blocking, write is busy waiting until TX FIFO has enough space. These functions are used by default.

Header File

- components/vfs/include/esp_vfs_eventfd.h

Functions

esp_err_t **esp_vfs_eventfd_register**(const *esp_vfs_eventfd_config_t* *config)

Registers the event vfs.

Returns ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered.

esp_err_t **esp_vfs_eventfd_unregister**(void)

Unregisters the event vfs.

Returns ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for given prefix hasn't been registered

int **eventfd**(unsigned int initval, int flags)

Structures

struct **esp_vfs_eventfd_config_t**

Eventfd vfs initialization settings.

Public Members

size_t **max_fds**

The maxinum number of eventfds supported

Macros

EFD_SUPPORT_ISR

ESP_VFS_EVENTD_CONFIG_DEFAULT()

2.9.9 Wear Levelling API

Overview

Most of flash memory and especially SPI flash that is used in ESP32 has a sector-based organization and also has a limited number of erase/modification cycles per memory sector. The wear levelling component helps to distribute wear and tear among sectors more evenly without requiring any attention from the user.

The wear levelling component provides API functions related to reading, writing, erasing, and memory mapping of data in external SPI flash through the partition component. The component also has higher-level API functions which work with the FAT filesystem defined in *FAT filesystem*.

The wear levelling component, together with the FAT FS component, uses FAT FS sectors of 4096 bytes, which is a standard size for flash memory. With this size, the component shows the best performance but needs additional memory in RAM.

To save internal memory, the component has two additional modes which both use sectors of 512 bytes:

- **Performance mode.** Erase sector operation data is stored in RAM, the sector is erased, and then data is copied back to flash memory. However, if a device is powered off for any reason, all 4096 bytes of data is lost.
- **Safety mode.** The data is first saved to flash memory, and after the sector is erased, the data is saved back. If a device is powered off, the data can be recovered as soon as the device boots up.

The default settings are as follows:

- Sector size is 512 bytes
- Performance mode

You can change the settings through the configuration menu.

The wear levelling component does not cache data in RAM. The write and erase functions modify flash directly, and flash contents are consistent when the function returns.

Wear Levelling access API functions

This is the set of API functions for working with data in flash:

- `wl_mount` - initializes the wear levelling module and mounts the specified partition
- `wl_unmount` - unmounts the partition and deinitializes the wear levelling module
- `wl_erase_range` - erases a range of addresses in flash
- `wl_write` - writes data to a partition
- `wl_read` - reads data from a partition
- `wl_size` - returns the size of available memory in bytes
- `wl_sector_size` - returns the size of one sector

As a rule, try to avoid using raw wear levelling functions and use filesystem-specific functions instead.

Memory Size

The memory size is calculated in the wear levelling module based on partition parameters. The module uses some sectors of flash for internal data.

See also

- *FAT Filesystem Support*
- *Partition Tables*

Application Example

An example that combines the wear levelling driver with the FATFS library is provided in the storage/wear_levelling directory. This example initializes the wear levelling driver, mounts FatFs partition, as well as writes and reads data from it using POSIX and C library APIs. See storage/wear_levelling/README.md for more information.

High-level API Reference

Header Files

- fatfs/vfs/esp_vfs_fat.h

High-level wear levelling functions `esp_vfs_fat_spiflash_mount_rw_wl()`, `esp_vfs_fat_spiflash_unmount_rw_wl()` and struct `esp_vfs_fat_mount_config_t` are described in *FAT Filesystem Support*.

Mid-level API Reference

Header File

- components/wear_levelling/include/wear_levelling.h

Functions

esp_err_t **wl_mount**(const *esp_partition_t* *partition, *wl_handle_t* *out_handle)

Mount WL for defined partition.

Parameters
- **partition** –that will be used for access
- **out_handle** –handle of the WL instance

Returns
- ESP_OK, if the allocation was successfully;
- ESP_ERR_INVALID_ARG, if WL allocation was unsuccessful;
- ESP_ERR_NO_MEM, if there was no memory to allocate WL components;

esp_err_t **wl_unmount**(*wl_handle_t* handle)

Unmount WL for defined partition.

Parameters **handle** –WL partition handle

Returns
- ESP_OK, if the operation completed successfully;
- or one of error codes from lower-level flash driver.

esp_err_t **wl_erase_range**(*wl_handle_t* handle, size_t start_addr, size_t size)

Erase part of the WL storage.

Parameters
- **handle** –WL handle that are related to the partition
- **start_addr** –Address where erase operation should start. Must be aligned to the result of function wl_sector_size(⋯).
- **size** –Size of the range which should be erased, in bytes. Must be divisible by result of function wl_sector_size(⋯)..

Returns
- ESP_OK, if the range was erased successfully;
- ESP_ERR_INVALID_ARG, if iterator or dst are NULL;

- ESP_ERR_INVALID_SIZE, if erase would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

esp_err_t **wl_write**(*wl_handle_t* handle, size_t dest_addr, const void *src, size_t size)

Write data to the WL storage.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using wl_erase_range function.

Note: Prior to writing to WL storage, make sure it has been erased with wl_erase_range call.

Parameters

- **handle** –WL handle that are related to the partition
- **dest_addr** –Address where the data should be written, relative to the beginning of the partition.
- **src** –Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- **size** –Size of data to be written, in bytes.

Returns

- ESP_OK, if data was written successfully;
- ESP_ERR_INVALID_ARG, if dst_offset exceeds partition size;
- ESP_ERR_INVALID_SIZE, if write would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

esp_err_t **wl_read**(*wl_handle_t* handle, size_t src_addr, void *dest, size_t size)

Read data from the WL storage.

Parameters

- **handle** –WL module instance that was initialized before
- **dest** –Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- **src_addr** –Address of the data to be read, relative to the beginning of the partition.
- **size** –Size of data to be read, in bytes.

Returns

- ESP_OK, if data was read successfully;
- ESP_ERR_INVALID_ARG, if src_offset exceeds partition size;
- ESP_ERR_INVALID_SIZE, if read would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

size_t **wl_size**(*wl_handle_t* handle)

Get size of the WL storage.

Parameters **handle** –WL module handle that was initialized before
Returns usable size, in bytes

size_t **wl_sector_size**(*wl_handle_t* handle)

Get sector size of the WL instance.

Parameters **handle** –WL module handle that was initialized before
Returns sector size, in bytes

Macros

WL_INVALID_HANDLE

Type Definitions

typedef int32_t **wl_handle_t**

wear levelling handle

Code examples for this API section are provided in the storage directory of ESP-IDF examples.

2.10 System API

2.10.1 App Image Format

An application image consists of the following structures:

1. The *esp_image_header_t* structure describes the mode of SPI flash and the count of memory segments.
2. The *esp_image_segment_header_t* structure describes each segment, its length, and its location in ESP32's memory, followed by the data with a length of `data_len`. The data offset for each segment in the image is calculated in the following way:

- offset for 0 Segment = sizeof(*esp_image_header_t*) + sizeof(*esp_image_segment_header_t*).
- offset for 1 Segment = offset for 0 Segment + length of 0 Segment + sizeof(*esp_image_segment_header_t*).
- offset for 2 Segment = offset for 1 Segment + length of 1 Segment + sizeof(*esp_image_segment_header_t*).
- ...

The count of each segment is defined in the `segment_count` field that is stored in *esp_image_header_t*. The count cannot be more than *ESP_IMAGE_MAX_SEGMENTS*.

To get the list of your image segments, please run the following command:

```
esptool.py --chip esp32 image_info build/app.bin
```

```
esptool.py v2.3.1
Image version: 1
Entry point: 40080ea4
13 segments
Segment 1: len 0x13ce0 load 0x3f400020 file_offs 0x00000018 SOC_DROM
Segment 2: len 0x00000 load 0x3ff80000 file_offs 0x00013d00 SOC_RTC_DRAM
Segment 3: len 0x00000 load 0x3ff80000 file_offs 0x00013d08 SOC_RTC_DRAM
Segment 4: len 0x028e0 load 0x3ffb0000 file_offs 0x00013d10 DRAM
Segment 5: len 0x00000 load 0x3ffb28e0 file_offs 0x000165f8 DRAM
Segment 6: len 0x00400 load 0x40080000 file_offs 0x00016600 SOC_IRAM
Segment 7: len 0x09600 load 0x40080400 file_offs 0x00016a08 SOC_IRAM
Segment 8: len 0x62e4c load 0x400d0018 file_offs 0x00020010 SOC_IROM
Segment 9: len 0x06cec load 0x40089a00 file_offs 0x00082e64 SOC_IROM
Segment 10: len 0x00000 load 0x400c0000 file_offs 0x00089b58 SOC_RTC_IRAM
Segment 11: len 0x00004 load 0x50000000 file_offs 0x00089b60 SOC_RTC_DATA
Segment 12: len 0x00000 load 0x50000004 file_offs 0x00089b6c SOC_RTC_DATA
Segment 13: len 0x00000 load 0x50000004 file_offs 0x00089b74 SOC_RTC_DATA
Checksum: e8 (valid)Validation Hash:␣
↪407089ca0eae2bbf83b4120979d3354b1c938a49cb7a0c997f240474ef2ec76b (valid)
```

You can also see the information on segments in the ESP-IDF logs while your application is booting:

```
I (443) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x13ce0 (␣
↪81120) map
I (489) esp_image: segment 1: paddr=0x00033d08 vaddr=0x3ff80000 size=0x00000 ( 0)␣
↪load
I (530) esp_image: segment 2: paddr=0x00033d10 vaddr=0x3ff80000 size=0x00000 ( 0)␣
↪load
```

(continues on next page)

(continued from previous page)

```
I (571) esp_image: segment 3: paddr=0x00033d18 vaddr=0x3ffb0000 size=0x028e0 (␣
↪10464) load
I (612) esp_image: segment 4: paddr=0x00036600 vaddr=0x3ffb28e0 size=0x00000 ( 0)␣
↪load
I (654) esp_image: segment 5: paddr=0x00036608 vaddr=0x40080000 size=0x00400 (␣
↪1024) load
I (695) esp_image: segment 6: paddr=0x00036a10 vaddr=0x40080400 size=0x09600 (␣
↪38400) load
I (737) esp_image: segment 7: paddr=0x00040018 vaddr=0x400d0018 size=0x62e4c␣
↪(405068) map
I (847) esp_image: segment 8: paddr=0x000a2e6c vaddr=0x40089a00 size=0x06cec (␣
↪27884) load
I (888) esp_image: segment 9: paddr=0x000a9b60 vaddr=0x400c0000 size=0x00000 ( 0)␣
↪load
I (929) esp_image: segment 10: paddr=0x000a9b68 vaddr=0x50000000 size=0x00004 ( 4)␣
↪load
I (971) esp_image: segment 11: paddr=0x000a9b74 vaddr=0x50000004 size=0x00000 ( 0)␣
↪load
I (1012) esp_image: segment 12: paddr=0x000a9b7c vaddr=0x50000004 size=0x00000 (␣
↪0) load
```

For more details on the type of memory segments and their address ranges, see *ESP32 Technical Reference Manual* > *System and Memory* > *Embedded Memory* [PDF].

3. The image has a single checksum byte after the last segment. This byte is written on a sixteen byte padded boundary, so the application image might need padding.
4. If the `hash_appended` field from *`esp_image_header_t`* is set then a SHA256 checksum will be appended. The value of SHA256 is calculated on the range from the first byte and up to this field. The length of this field is 32 bytes.
5. If the options *CONFIG_SECURE_SIGNED_APPS_SCHEME* is set to ECDSA then the application image will have additional 68 bytes for an ECDSA signature, which includes:

- version word (4 bytes),
- signature data (64 bytes).

Application Description

The `DROM` segment starts with the *`esp_app_desc_t`* structure which carries specific fields describing the application:

- `magic_word` - the magic word for the esp_app_desc structure.
- `secure_version` - see *Anti-rollback*.
- `version` - see *App version*. *
- `project_name` is filled from `PROJECT_NAME`. *
- `time` and `date` - compile time and date.
- `idf_ver` - version of ESP-IDF. *
- `app_elf_sha256` - contains sha256 for the elf application file.

* - The maximum length is 32 characters, including null-termination character. For example, if the length of `PROJECT_NAME` exceeds 32 characters, the excess characters will be disregarded.

This structure is useful for identification of images uploaded OTA because it has a fixed offset = sizeof(*`esp_image_header_t`*) + sizeof(*`esp_image_segment_header_t`*). As soon as a device receives the first fragment containing this structure, it has all the information to determine whether the update should be continued or not.

Adding a Custom Structure to an Application

Users also have the opportunity to have similar structure with a fixed offset relative to the beginning of the image. The following pattern can be used to add a custom structure to your image:

```
const __attribute__((section(".rodata_custom_desc"))) esp_custom_app_desc_t custom_
↪app_desc = { ... }
```

Offset for custom structure is sizeof(*esp_image_header_t*) + sizeof(*esp_image_segment_header_t*) + sizeof(*esp_app_desc_t*).

To guarantee that the custom structure is located in the image even if it is not used, you need to add `target_link_libraries(${COMPONENT_TARGET} "-u custom_app_desc")` into `CMakeLists.txt`.

API Reference

Header File

- components/bootloader_support/include/esp_app_format.h

Structures

struct **esp_image_header_t**

Main header of binary image.

Public Members

uint8_t **magic**

Magic word ESP_IMAGE_HEADER_MAGIC

uint8_t **segment_count**

Count of memory segments

uint8_t **spi_mode**

flash read mode (esp_image_spi_mode_t as uint8_t)

uint8_t **spi_speed**

flash frequency (esp_image_spi_freq_t as uint8_t)

uint8_t **spi_size**

flash chip size (esp_image_flash_size_t as uint8_t)

uint32_t **entry_addr**

Entry address

uint8_t **wp_pin**

WP pin when SPI pins set via efuse (read by ROM bootloader, the IDF bootloader uses software to configure the WP pin and sets this field to 0xEE=disabled)

uint8_t **spi_pin_drv**[3]

Drive settings for the SPI flash pins (read by ROM bootloader)

esp_chip_id_t **chip_id**

Chip identification number

uint8_t **min_chip_rev**

Minimal chip revision supported by image After the Major and Minor revision eFuses were introduced into the chips, this field is no longer used. But for compatibility reasons, we keep this field and the data in it. Use min_chip_rev_full instead. The software interprets this as a Major version for most of the chips and as a Minor version for the ESP32-C3.

uint16_t **min_chip_rev_full**

Minimal chip revision supported by image, in format: major * 100 + minor

uint16_t **max_chip_rev_full**

Maximal chip revision supported by image, in format: major * 100 + minor

uint8_t **reserved**[4]

Reserved bytes in additional header space, currently unused

uint8_t **hash_appended**

If 1, a SHA256 digest "simple hash" (of the entire image) is appended after the checksum. Included in image length. This digest is separate to secure boot and only used for detecting corruption. For secure boot signed images, the signature is appended after this (and the simple hash is included in the signed data).

struct **esp_image_segment_header_t**

Header of binary image segment.

Public Members

uint32_t **load_addr**

Address of segment

uint32_t **data_len**

Length of data

Macros

ESP_IMAGE_HEADER_MAGIC

The magic word for the *esp_image_header_t* structure.

ESP_IMAGE_MAX_SEGMENTS

Max count of segments in the image.

Enumerations

enum **esp_chip_id_t**

ESP chip ID.

Values:

enumerator **ESP_CHIP_ID_ESP32**

chip ID: ESP32

enumerator **ESP_CHIP_ID_ESP32S2**
chip ID: ESP32-S2

enumerator **ESP_CHIP_ID_ESP32C3**
chip ID: ESP32-C3

enumerator **ESP_CHIP_ID_ESP32S3**
chip ID: ESP32-S3

enumerator **ESP_CHIP_ID_ESP32C2**
chip ID: ESP32-C2

enumerator **ESP_CHIP_ID_INVALID**
Invalid chip ID (we defined it to make sure the esp_chip_id_t is 2 bytes size)

enum **esp_image_spi_mode_t**
SPI flash mode, used in *esp_image_header_t*.

Values:

enumerator **ESP_IMAGE_SPI_MODE_QIO**
SPI mode QIO

enumerator **ESP_IMAGE_SPI_MODE_QOUT**
SPI mode QOUT

enumerator **ESP_IMAGE_SPI_MODE_DIO**
SPI mode DIO

enumerator **ESP_IMAGE_SPI_MODE_DOUT**
SPI mode DOUT

enumerator **ESP_IMAGE_SPI_MODE_FAST_READ**
SPI mode FAST_READ

enumerator **ESP_IMAGE_SPI_MODE_SLOW_READ**
SPI mode SLOW_READ

enum **esp_image_spi_freq_t**
SPI flash clock division factor.

Values:

enumerator **ESP_IMAGE_SPI_SPEED_DIV_2**
The SPI flash clock frequency is divided by 2 of the clock source

enumerator **ESP_IMAGE_SPI_SPEED_DIV_3**
The SPI flash clock frequency is divided by 3 of the clock source

enumerator **ESP_IMAGE_SPI_SPEED_DIV_4**
The SPI flash clock frequency is divided by 4 of the clock source

enumerator **ESP_IMAGE_SPI_SPEED_DIV_1**

The SPI flash clock frequency equals to the clock source

enum **esp_image_flash_size_t**

Supported SPI flash sizes.

Values:

enumerator **ESP_IMAGE_FLASH_SIZE_1MB**

SPI flash size 1 MB

enumerator **ESP_IMAGE_FLASH_SIZE_2MB**

SPI flash size 2 MB

enumerator **ESP_IMAGE_FLASH_SIZE_4MB**

SPI flash size 4 MB

enumerator **ESP_IMAGE_FLASH_SIZE_8MB**

SPI flash size 8 MB

enumerator **ESP_IMAGE_FLASH_SIZE_16MB**

SPI flash size 16 MB

enumerator **ESP_IMAGE_FLASH_SIZE_32MB**

SPI flash size 32 MB

enumerator **ESP_IMAGE_FLASH_SIZE_64MB**

SPI flash size 64 MB

enumerator **ESP_IMAGE_FLASH_SIZE_128MB**

SPI flash size 128 MB

enumerator **ESP_IMAGE_FLASH_SIZE_MAX**

SPI flash size MAX

2.10.2 Application Level Tracing

Overview

IDF provides a useful feature for program behavior analysis called **Application Level Tracing**. The feature can be enabled in menuconfig and allows transfer of arbitrary data between the host and ESP32 via JTAG interface with minimal overhead on program execution. Developers can use this library to send application specific state of execution to the host and receive commands or other type of info in the opposite direction at runtime. The main use cases of this library are:

1. Collecting application specific data, see *Application Specific Tracing*
2. Lightweight logging to the host, see *Logging to Host*
3. System behaviour analysis, see *System Behavior Analysis with SEGGER SystemView*

API Reference

Header File

- components/app_trace/include/esp_app_trace.h

Functions

esp_err_t **esp_apptrace_init**(void)

Initializes application tracing module.

Note: Should be called before any esp_apptrace_xxx call.

Returns ESP_OK on success, otherwise see esp_err_t

void **esp_apptrace_down_buffer_config**(uint8_t *buf, uint32_t size)

Configures down buffer.

Note: Needs to be called before attempting to receive any data using esp_apptrace_down_buffer_get and esp_apptrace_read. This function does not protect internal data by lock.

Parameters

- **buf** –Address of buffer to use for down channel (host to target) data.
- **size** –Size of the buffer.

uint8_t ***esp_apptrace_buffer_get**(*esp_apptrace_dest_t* dest, uint32_t size, uint32_t tmo)

Allocates buffer for trace data. Once the data in the buffer is ready to be sent, esp_apptrace_buffer_put must be called to indicate it.

Parameters

- **dest** –Indicates HW interface to send data.
- **size** –Size of data to write to trace buffer.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns non-NULL on success, otherwise NULL.

esp_err_t **esp_apptrace_buffer_put**(*esp_apptrace_dest_t* dest, uint8_t *ptr, uint32_t tmo)

Indicates that the data in the buffer is ready to be sent. This function is a counterpart of and must be preceded by esp_apptrace_buffer_get.

Parameters

- **dest** –Indicates HW interface to send data. Should be identical to the same parameter in call to esp_apptrace_buffer_get.
- **ptr** –Address of trace buffer to release. Should be the value returned by call to esp_apptrace_buffer_get.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

esp_err_t **esp_apptrace_write**(*esp_apptrace_dest_t* dest, const void *data, uint32_t size, uint32_t tmo)

Writes data to trace buffer.

Parameters

- **dest** –Indicates HW interface to send data.
- **data** –Address of data to write to trace buffer.
- **size** –Size of data to write to trace buffer.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

int **esp_apptrace_vprintf_to**(*esp_apptrace_dest_t* dest, uint32_t tmo, const char *fmt, va_list ap)

vprintf-like function to send log messages to host via specified HW interface.

Parameters
- **dest** –Indicates HW interface to send data.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.
- **fmt** –Address of format string.
- **ap** –List of arguments.

Returns Number of bytes written.

int **esp_apptrace_vprintf**(const char *fmt, va_list ap)

vprintf-like function to send log messages to host.

Parameters
- **fmt** –Address of format string.
- **ap** –List of arguments.

Returns Number of bytes written.

esp_err_t **esp_apptrace_flush**(*esp_apptrace_dest_t* dest, uint32_t tmo)

Flushes remaining data in trace buffer to host.

Parameters
- **dest** –Indicates HW interface to flush data on.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

esp_err_t **esp_apptrace_flush_nolock**(*esp_apptrace_dest_t* dest, uint32_t min_sz, uint32_t tmo)

Flushes remaining data in trace buffer to host without locking internal data. This is a special version of esp_apptrace_flush which should be called from panic handler.

Parameters
- **dest** –Indicates HW interface to flush data on.
- **min_sz** –Threshold for flushing data. If current filling level is above this value, data will be flushed. TRAX destinations only.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

esp_err_t **esp_apptrace_read**(*esp_apptrace_dest_t* dest, void *data, uint32_t *size, uint32_t tmo)

Reads host data from trace buffer.

Parameters
- **dest** –Indicates HW interface to read the data on.
- **data** –Address of buffer to put data from trace buffer.
- **size** –Pointer to store size of read data. Before call to this function pointed memory must hold requested size of data
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

uint8_t ***esp_apptrace_down_buffer_get**(*esp_apptrace_dest_t* dest, uint32_t *size, uint32_t tmo)

Retrieves incoming data buffer if any. Once data in the buffer is processed, esp_apptrace_down_buffer_put must be called to indicate it.

Parameters
- **dest** –Indicates HW interface to receive data.
- **size** –Address to store size of available data in down buffer. Must be initialized with requested value.

- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns non-NULL on success, otherwise NULL.

esp_err_t **esp_apptrace_down_buffer_put** (*esp_apptrace_dest_t* dest, uint8_t *ptr, uint32_t tmo)

Indicates that the data in the down buffer is processed. This function is a counterpart of and must be preceded by esp_apptrace_down_buffer_get.

Parameters

- **dest** –Indicates HW interface to receive data. Should be identical to the same parameter in call to esp_apptrace_down_buffer_get.
- **ptr** –Address of trace buffer to release. Should be the value returned by call to esp_apptrace_down_buffer_get.
- **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

bool **esp_apptrace_host_is_connected** (*esp_apptrace_dest_t* dest)

Checks whether host is connected.

Parameters **dest** –Indicates HW interface to use.

Returns true if host is connected, otherwise false

void ***esp_apptrace_fopen** (*esp_apptrace_dest_t* dest, const char *path, const char *mode)

Opens file on host. This function has the same semantic as ‘fopen’ except for the first argument.

Parameters

- **dest** –Indicates HW interface to use.
- **path** –Path to file.
- **mode** –Mode string. See fopen for details.

Returns non zero file handle on success, otherwise 0

int **esp_apptrace_fclose** (*esp_apptrace_dest_t* dest, void *stream)

Closes file on host. This function has the same semantic as ‘fclose’ except for the first argument.

Parameters

- **dest** –Indicates HW interface to use.
- **stream** –File handle returned by esp_apptrace_fopen.

Returns Zero on success, otherwise non-zero. See fclose for details.

size_t **esp_apptrace_fwrite** (*esp_apptrace_dest_t* dest, const void *ptr, size_t size, size_t nmemb, void *stream)

Writes to file on host. This function has the same semantic as ‘fwrite’ except for the first argument.

Parameters

- **dest** –Indicates HW interface to use.
- **ptr** –Address of data to write.
- **size** –Size of an item.
- **nmemb** –Number of items to write.
- **stream** –File handle returned by esp_apptrace_fopen.

Returns Number of written items. See fwrite for details.

size_t **esp_apptrace_fread** (*esp_apptrace_dest_t* dest, void *ptr, size_t size, size_t nmemb, void *stream)

Read file on host. This function has the same semantic as ‘fread’ except for the first argument.

Parameters

- **dest** –Indicates HW interface to use.
- **ptr** –Address to store read data.
- **size** –Size of an item.
- **nmemb** –Number of items to read.
- **stream** –File handle returned by esp_apptrace_fopen.

Returns Number of read items. See fread for details.

int **esp_apptrace_fseek**(*esp_apptrace_dest_t* dest, void *stream, long offset, int whence)

Set position indicator in file on host. This function has the same semantic as 'fseek' except for the first argument.

Parameters
- **dest** –Indicates HW interface to use.
- **stream** –File handle returned by esp_apptrace_fopen.
- **offset** –Offset. See fseek for details.
- **whence** –Position in file. See fseek for details.

Returns Zero on success, otherwise non-zero. See fseek for details.

int **esp_apptrace_ftell**(*esp_apptrace_dest_t* dest, void *stream)

Get current position indicator for file on host. This function has the same semantic as 'ftell' except for the first argument.

Parameters
- **dest** –Indicates HW interface to use.
- **stream** –File handle returned by esp_apptrace_fopen.

Returns Current position in file. See ftell for details.

int **esp_apptrace_fstop**(*esp_apptrace_dest_t* dest)

Indicates to the host that all file operations are complete. This function should be called after all file operations are finished and indicate to the host that it can perform cleanup operations (close open files etc.).

Parameters **dest** –Indicates HW interface to use.
Returns ESP_OK on success, otherwise see esp_err_t

void **esp_gcov_dump**(void)

Triggers gcov info dump. This function waits for the host to connect to target before dumping data.

Enumerations

enum **esp_apptrace_dest_t**

Application trace data destinations bits.

Values:

enumerator **ESP_APPTRACE_DEST_JTAG**

JTAG destination.

enumerator **ESP_APPTRACE_DEST_TRAX**

xxx_TRAX name is obsolete, use more common xxx_JTAG

enumerator **ESP_APPTRACE_DEST_UART**

UART destination.

enumerator **ESP_APPTRACE_DEST_MAX**

enumerator **ESP_APPTRACE_DEST_NUM**

Header File

- components/app_trace/include/esp_sysview_trace.h

Functions

static inline *esp_err_t* **esp_sysview_flush**(uint32_t tmo)

Flushes remaining data in SystemView trace buffer to host.

Parameters **tmo** –Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

Returns ESP_OK.

int **esp_sysview_vprintf**(const char *format, va_list args)

vprintf-like function to sent log messages to the host.

Parameters

- **format** –Address of format string.
- **args** –List of arguments.

Returns Number of bytes written.

esp_err_t **esp_sysview_heap_trace_start**(uint32_t tmo)

Starts SystemView heap tracing.

Parameters **tmo** –Timeout (in us) to wait for the host to be connected. Use -1 to wait forever.

Returns ESP_OK on success, ESP_ERR_TIMEOUT if operation has been timed out.

esp_err_t **esp_sysview_heap_trace_stop**(void)

Stops SystemView heap tracing.

Returns ESP_OK.

void **esp_sysview_heap_trace_alloc**(void *addr, uint32_t size, const void *callers)

Sends heap allocation event to the host.

Parameters

- **addr** –Address of allocated block.
- **size** –Size of allocated block.
- **callers** –Pointer to array with callstack addresses. Array size must be CONFIG_HEAP_TRACING_STACK_DEPTH.

void **esp_sysview_heap_trace_free**(void *addr, const void *callers)

Sends heap de-allocation event to the host.

Parameters

- **addr** –Address of de-allocated block.
- **callers** –Pointer to array with callstack addresses. Array size must be CONFIG_HEAP_TRACING_STACK_DEPTH.

2.10.3 Call function with external stack

Overview

A given function can be executed with a user allocated stack space which is independent of current task stack, this mechanism can be used to save stack space wasted by tasks which call a common function with intensive stack usage such as *printf*. The given function can be called inside the shared stack space which is a callback function deferred by calling `esp_execute_shared_stack_function()`, passing that function as parameter.

Usage

`esp_execute_shared_stack_function()` takes four arguments:

- a mutex object allocated by the caller, which is used to protect if the same function shares its allocated stack
- a pointer to the top of stack used for that fuction
- the size of stack in bytes
- a pointer to the shared stack function

The user defined function will be deferred as a callback and can be called using the user allocated space without taking space from current task stack.

The usage may look like the code below:

```
void external_stack_function(void)
{
    printf("Executing this printf from external stack! \n");
}

//Let's suppose we want to call printf using a separated stack space
//allowing the app to reduce its stack size.
void app_main()
{
    //Allocate a stack buffer, from heap or as a static form:
    portSTACK_TYPE *shared_stack = malloc(8192 * sizeof(portSTACK_TYPE));
    assert(shared_stack != NULL);

    //Allocate a mutex to protect its usage:
    SemaphoreHandle_t printf_lock = xSemaphoreCreateMutex();
    assert(printf_lock != NULL);

    //Call the desired function using the macro helper:
    esp_execute_shared_stack_function(printf_lock,
                                      shared_stack,
                                      8192,
                                      external_stack_function);

    vSemaphoreDelete(printf_lock);
    free(shared_stack);
}
```

API Reference

Header File

- components/esp_system/include/esp_expression_with_stack.h

Functions

void **esp_execute_shared_stack_function**(*SemaphoreHandle_t* lock, void *stack, size_t stack_size, *shared_stack_function* function)

Calls user defined shared stack space function.

Note: if either lock, stack or stack size is invalid, the expression will be called using the current stack.

Parameters

- **lock** –Mutex object to protect in case of shared stack
- **stack** –Pointer to user alocated stack
- **stack_size** –Size of current stack in bytes
- **function** –pointer to the shared stack function to be executed

Macros

ESP_EXECUTE_EXPRESSION_WITH_STACK(lock, stack, stack_size, expression)

Type Definitions

typedef void (*`shared_stack_function`)(void)

2.10.4 Chip Revision

Overview

A new chip versioning logic was introduced in new chips. Chips have several eFuse version fields:

- Major wafer version (`WAFER_VERSION_MAJOR` eFuse)
- Minor wafer version (`WAFER_VERSION_MINOR` eFuse)
- Ignore maximal revision (`DISABLE_WAFER_VERSION_MAJOR` eFuse)

The new versioning logic is being introduced to distinguish changes in chips as breaking changes and non-breaking changes. Chips with non-breaking changes can run the same software as the previous chip. The previous chip means that the major version is the same.

If the newly released chip does not have breaking changes, that means it can run the same software as the previous chip, then in that chip we keep the same major version and increment the minor version by 1. Otherwise, if there is a breaking change in the newly released chip, meaning it can not run the same software as the previous chip, then in that chip we increase the major version and set the minor version to 0.

The software supports a number of revisions, from the minimum to the maximum (the min/max configs are defined in Kconfig). If the software is unaware of a new chip (when the chip version is out of range), it will refuse to run on it unless the Ignore maximum revision restrictions bit is set. This bit removes the upper revision limit.

Minimum versions limits the software to only run on a chip revision that is high enough to support some features. Maximum version is the maximum version that is well-supported by current software. When chip version is above the maximum version, software will reject to boot, because it may not work on, or work with risk on the chip.

Adding the major and minor wafer revision make the versioning logic is branchable.

Note: The previous versioning logic was based on a single eFuse version field (`WAFER_VERSION`). This approach makes it impossible to mark chips as breaking or non-breaking changes, and the versioning logic becomes linear.

Using the branched versioning scheme allows us to support more chips in the software without updating the software when a new released compatible chip is used. Thus, the software will be compatible with as many new chip revisions as possible. If the software is no longer compatible with a new chip with breaking changes, the software will abort.

Revisions

ECO	Revision (Major.Minor)
ECO0	v0.0
ECO1	v1.0
ECO1.1	v1.1
ECO2	v2.0
ECO3	v3.0
ECO4	v3.1

Chip Revision `vX.Y`, where:

- `X` means Major wafer version. If it is changed, it means that the current software version is not compatible with this released chip and the software must be updated to use this chip.
- `Y` means Minor wafer version. If it is changed that means the current software version is compatible with the released chip, and there is no need to update the software.

The `vX.Y` chip version format will be used further instead of the ECO number.

Representing Revision Requirement Of A Binary Image

The 2nd stage bootloader and the application binary images have the *esp_image_header_t* header, which stores the revision numbers of the chip on which the software can be run. This header has 3 fields related to revisions:

- `min_chip_rev` - Minimal chip MAJOR revision required by image (but for ESP32-C3 it is MINOR revision). Its value is determined by *CONFIG_ESP32_REV_MIN*.
- `min_chip_rev_full` - Minimal chip MINOR revision required by image in format: `major * 100 + minor`. Its value is determined by *CONFIG_ESP32_REV_MIN*.
- `max_chip_rev_full` - Maximal chip revision required by image in format: `major * 100 + minor`. Its value is determined by `CONFIG_ESP32_REV_MAX_FULL`. It can not be changed by user. Only Espressif can change it when a new version will be supported in IDF.

Chip Revision APIs

These APIs helps to get chip revision from eFuses:

- *efuse_hal_chip_revision()*. It returns revision in the `major * 100 + minor` format.
- *efuse_hal_get_major_chip_version()*. It returns Major revision.
- *efuse_hal_get_minor_chip_version()*. It returns Minor revision.

The following Kconfig definitions (in `major * 100 + minor` format) that can help add the chip revision dependency to the code:

- `CONFIG_ESP32_REV_MIN_FULL`
- `CONFIG_ESP_REV_MIN_FULL`
- `CONFIG_ESP32_REV_MAX_FULL`
- `CONFIG_ESP_REV_MAX_FULL`

Maximal And Minimal Revision Restrictions

The order for checking the minimum and maximum revisions:

1. The 1st stage bootloader (ROM bootloader) does not check minimal and maximal revision fields from *esp_image_header_t* before running the 2nd stage bootloader.
2. The 2nd stage bootloader checks at the initialization phase that bootloader itself can be launched on the chip of this revision. It extracts the minimum revision from the header of the bootloader image and checks against the chip revision from eFuses. If the chip revision is less than the minimum revision, the bootloader refuses to boot up and aborts. The maximum revision is not checked at this phase.
3. Then the 2nd stage bootloader checks the revision requirements of the application. It extracts the minimum and maximum revisions from the header of the application image and checks against the chip revision from eFuses. If the chip revision is less than the minimum revision or higher than the maximum revision, the bootloader refuses to boot up and aborts. However, if the Ignore maximal revision bit is set, the maximum revision constraint can be ignored. The ignore bit is set by the customer themself when there is confirmation that the software is able to work with this chip revision.
4. Further, at the OTA update stage, the running application checks if the new software matches the chip revision. It extracts the minimum and maximum revisions from the header of the new application image and checks against the chip revision from eFuses. It checks for revision matching in the same way that the bootloader does, so that the chip revision is between the min and max revisions (logic of ignoring max revision also applies).

Issues

1. If the 2nd stage bootloader is run on the chip revision < minimum revision shown in the image, a reboot occurs. The following message will be printed:

```
Image requires chip rev >= v3.0, but chip is v1.0
```

To resolve this issue:

- make sure the chip you are using is suitable for the software, or use a chip with the required minimum revision or higher.
- update the software with *CONFIG_ESP32_REV_MIN* to get it <= the revision of chip being used

2. If application does not match minimal and maximal chip revisions, a reboot occurs. The following message will be printed:

```
Image requires chip rev <= v2.99, but chip is v3.0
```

To resolve this issue, update the IDF to a newer version that supports the used chip (`CONFIG_ESP32_REV_MAX_FULL`). Another way to fix this is to set the `Ignore maximal revision` bit in eFuse or use a chip that is suitable for the software.

Backward Compatible With Bootloaders Built By Older ESP-IDF Versions

The old bootloaders (IDF < 5.0) do not know about Major and Minor wafer version eFuses. They use one single eFuse for this - wafer version.

The old bootloaders did not read the minor wafer version eFuse, the major version can be only <= 3. So it means that the old bootloader can detect correctly only chip version in range v0.0 - v3.0, where the minor version is always 0.

Please check the chip version using `esptool chip_id` command.

API Reference

Header File

- components/hal/include/hal/efuse_hal.h

Functions

void **efuse_hal_get_mac**(uint8_t *mac)

get factory mac address

uint32_t **efuse_hal_chip_revision**(void)

Returns chip version.

Returns Chip version in format: Major * 100 + Minor

bool **efuse_hal_flash_encryption_enabled**(void)

Is flash encryption currently enabled in hardware?

Flash encryption is enabled if the FLASH_CRYPT_CNT efuse has an odd number of bits set.

Returns true if flash encryption is enabled.

uint32_t **efuse_hal_get_major_chip_version**(void)

Returns major chip version.

uint32_t **efuse_hal_get_minor_chip_version**(void)

Returns minor chip version.

2.10.5 Console

ESP-IDF provides `console` component, which includes building blocks needed to develop an interactive console over serial port. This component includes following facilities:

- Line editing, provided by linenoise library. This includes handling of backspace and arrow keys, scrolling through command history, command auto-completion, and argument hints.
- Splitting of command line into arguments.

- Argument parsing, provided by argtable3 library. This library includes APIs used for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.
- Functions to establish a basic REPL (Read-Evaluate-Print-Loop) environment.

Note: These facilities can be used together or independently. For example, it is possible to use line editing and command registration features, but use `getopt` or custom code for argument parsing, instead of argtable3. Likewise, it is possible to use simpler means of command input (such as `fgets`) together with the rest of the means for command splitting and argument parsing.

Line editing

Line editing feature lets users compose commands by typing them, erasing symbols using 'backspace' key, navigating within the command using left/right keys, navigating to previously typed commands using up/down keys, and performing autocompletion using 'tab' key.

Note: This feature relies on ANSI escape sequence support in the terminal application. As such, serial monitors which display raw UART data can not be used together with the line editing library. If you see `[6n` or similar escape sequence when running system/console example instead of a command prompt (e.g. `esp>`), it means that the serial monitor does not support escape sequences. Programs which are known to work are GNU screen, minicom, and idf_monitor.py (which can be invoked using `idf.py monitor` from project directory).

Here is an overview of functions provided by linenoise library.

Configuration Linenoise library does not need explicit initialization. However, some configuration defaults may need to be changed before invoking the main line editing function.

`linenoiseClearScreen()`

Clear terminal screen using an escape sequence and position the cursor at the top left corner.

`linenoiseSetMultiLine()`

Switch between single line and multi line editing modes. In single line mode, if the length of the command exceeds the width of the terminal, the command text is scrolled within the line to show the end of the text. In this case the beginning of the text is hidden. Single line needs less data to be sent to refresh screen on each key press, so exhibits less glitching compared to the multi line mode. On the flip side, editing commands and copying command text from terminal in single line mode is harder. Default is single line mode.

`linenoiseAllowEmpty()`

Set whether linenoise library will return a zero-length string (if `true`) or `NULL` (if `false`) for empty lines. By default, zero-length strings are returned.

`linenoiseSetMaxLineLen()`

Set maximum length of the line for linenoise library. Default length is 4096. If you need optimize RAM memory usage, you can do it by this function by setting a value less than default 4 KB.

Main loop `linenoise()`

In most cases, console applications have some form of read/eval loop. `linenoise()` is the single function which handles user's key presses and returns completed line once 'enter' key is pressed. As such, it handles the 'read' part of the loop.

`linenoiseFree()`

This function must be called to release the command line buffer obtained from `linenoise()` function.

Hints and completions `linenoiseSetCompletionCallback()`

When user presses 'tab' key, linenoise library invokes completion callback. The callback should inspect the contents of the command typed so far and provide a list of possible completions using calls to `linenoiseAddCompletion()` function. `linenoiseSetCompletionCallback()` function should be called to register this completion callback, if completion feature is desired.

`console` component provides a ready made function to provide completions for registered commands, *`esp_console_get_completion()`* (see below).

`linenoiseAddCompletion()`

Function to be called by completion callback to inform the library about possible completions of the currently typed command.

`linenoiseSetHintsCallback()`

Whenever user input changes, linenoise invokes hints callback. This callback can inspect the command line typed so far, and provide a string with hints (which can include list of command arguments, for example). The library then displays the hint text on the same line where editing happens, possibly with a different color.

`linenoiseSetFreeHintsCallback()`

If hint string returned by hints callback is dynamically allocated or needs to be otherwise recycled, the function which performs such cleanup should be registered via `linenoiseSetFreeHintsCallback()`.

History `linenoiseHistorySetMaxLen()`

This function sets the number of most recently typed commands to be kept in memory. Users can navigate the history using up/down arrows.

`linenoiseHistoryAdd()`

Linenoise does not automatically add commands to history. Instead, applications need to call this function to add command strings to the history.

`linenoiseHistorySave()`

Function saves command history from RAM to a text file, for example on an SD card or on a filesystem in flash memory.

`linenoiseHistoryLoad()`

Counterpart to `linenoiseHistorySave()`, loads history from a file.

`linenoiseHistoryFree()`

Releases memory used to store command history. Call this function when done working with linenoise library.

Splitting of command line into arguments

`console` component provides *`esp_console_split_argv()`* function to split command line string into arguments. The function returns the number of arguments found (`argc`) and fills an array of pointers which can be passed as `argv` argument to any function which accepts arguments in `argc, argv` format.

The command line is split into arguments according to the following rules:

- Arguments are separated by spaces
- If spaces within arguments are required, they can be escaped using `\` (backslash) character.
- Other escape sequences which are recognized are `\\` (which produces literal backslash) and `\"`, which produces a double quote.

- Arguments can be quoted using double quotes. Quotes may appear only in the beginning and at the end of the argument. Quotes within the argument must be escaped as mentioned above. Quotes surrounding the argument are stripped by `esp_console_split_argv` function.

Examples:

- `abc def 1 20 .3` → [`abc`, `def`, `1`, `20`, `.3`]
- `abc "123 456" def` → [`abc`, `123 456`, `def`]
- `` `a\ b\\c\" `` → [`a b\c"`]

Argument parsing

For argument parsing, `console` component includes argtable3 library. Please see tutorial for an introduction to argtable3. Github repository also includes examples.

Command registration and dispatching

`console` component includes utility functions which handle registration of commands, matching commands typed by the user to registered ones, and calling these commands with the arguments given on the command line.

Application first initializes command registration module using a call to *`esp_console_init()`*, and calls *`esp_console_cmd_register()`* function to register command handlers.

For each command, application provides the following information (in the form of `esp_console_cmd_t` structure):

- Command name (string without spaces)
- Help text explaining what the command does
- Optional hint text listing the arguments of the command. If application uses Argtable3 for argument parsing, hint text can be generated automatically by providing a pointer to argtable argument definitions structure instead.
- The command handler function.

A few other functions are provided by the command registration module:

`esp_console_run()`

> This function takes the command line string, splits it into argc/argv argument list using *`esp_console_split_argv()`*, looks up the command in the list of registered components, and if it is found, executes its handler.

`esp_console_register_help_command()`

> Adds `help` command to the list of registered commands. This command prints the list of all the registered commands, along with their arguments and help texts.

`esp_console_get_completion()`

> Callback function to be used with `linenoiseSetCompletionCallback()` from linenoise library. Provides completions to linenoise based on the list of registered commands.

`esp_console_get_hint()`

> Callback function to be used with `linenoiseSetHintsCallback()` from linenoise library. Provides argument hints for registered commands to linenoise.

Initialize console REPL environment

To establish a basic REPL environment, `console` component provides several useful APIs, combining those functions described above.

In a typical application, you only need to call *esp_console_new_repl_uart()* to initialize the REPL environment based on UART device, including driver install, basic console configuration, spawning a thread to do REPL task and register several useful commands (e.g. *help*).

After that, you can register your own commands with *esp_console_cmd_register()*. The REPL environment keeps in init state until you call *esp_console_start_repl()*.

Application Example

Example application illustrating usage of the `console` component is available in system/console directory. This example shows how to initialize UART and VFS functions, set up linenoise library, read and handle commands from UART, and store command history in Flash. See README.md in the example directory for more details.

Besides that, ESP-IDF contains several useful examples which based on *console* component and can be treated as "tools" when developing applications. For example, peripherals/i2c/i2c_tools, wifi/iperf.

API Reference

Header File

- components/console/esp_console.h

Functions

esp_err_t **esp_console_init** (const *esp_console_config_t* *config)

initialize console module

Note: Call this once before using other console module features

Parameters **config** –console configuration

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_STATE if already initialized
- ESP_ERR_INVALID_ARG if the configuration is invalid

esp_err_t **esp_console_deinit** (void)

de-initialize console module

Note: Call this once when done using console module functions

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized yet

esp_err_t **esp_console_cmd_register** (const *esp_console_cmd_t* *cmd)

Register console command.

Parameters **cmd** –pointer to the command description; can point to a temporary value

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_ARG if command description includes invalid arguments

esp_err_t **esp_console_run**(const char *cmdline, int *cmd_ret)

Run command line.

Parameters
- **cmdline** –command line (command name followed by a number of arguments)
- **cmd_ret** –**[out]** return code from the command (set if command was run)

Returns
- ESP_OK, if command was run
- ESP_ERR_INVALID_ARG, if the command line is empty, or only contained whitespace
- ESP_ERR_NOT_FOUND, if command with given name wasn't registered
- ESP_ERR_INVALID_STATE, if esp_console_init wasn't called

size_t **esp_console_split_argv**(char *line, char **argv, size_t argv_size)

Split command line into arguments in place.

```
* - This function finds whitespace-separated arguments in the given input line.
*
*     'abc def 1 20 .3' -> [ 'abc', 'def', '1', '20', '.3' ]
*
* - Argument which include spaces may be surrounded with quotes. In this case
*   spaces are preserved and quotes are stripped.
*
*     'abc "123 456" def' -> [ 'abc', '123 456', 'def' ]
*
* - Escape sequences may be used to produce backslash, double quote, and space:
*
*     'a\ b\\c\"' -> [ 'a b\c"' ]
*
```

Note: Pointers to at most argv_size - 1 arguments are returned in argv array. The pointer after the last one (i.e. argv[argc]) is set to NULL.

Parameters
- **line** –pointer to buffer to parse; it is modified in place
- **argv** –array where the pointers to arguments are written
- **argv_size** –number of elements in argv_array (max. number of arguments)

Returns number of arguments found (argc)

void **esp_console_get_completion**(const char *buf, *linenoiseCompletions* *lc)

Callback which provides command completion for linenoise library.

When using linenoise for line editing, command completion support can be enabled like this:

linenoiseSetCompletionCallback(&esp_console_get_completion);

Parameters
- **buf** –the string typed by the user
- **lc** –linenoiseCompletions to be filled in

const char ***esp_console_get_hint**(const char *buf, int *color, int *bold)

Callback which provides command hints for linenoise library.

When using linenoise for line editing, hints support can be enabled as follows:

linenoiseSetHintsCallback((linenoiseHintsCallback*) &esp_console_get_hint);

The extra cast is needed because linenoiseHintsCallback is defined as returning a char* instead of const char*.

Parameters
- **buf** –line typed by the user

- **color** –**[out]** ANSI color code to be used when displaying the hint
- **bold** –**[out]** set to 1 if hint has to be displayed in bold

Returns string containing the hint text. This string is persistent and should not be freed (i.e. linenoiseSetFreeHintsCallback should not be used).

esp_err_t **esp_console_register_help_command**(void)

Register a 'help' command.

Default 'help' command prints the list of registered commands along with hints and help strings.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE, if esp_console_init wasn't called

esp_err_t **esp_console_new_repl_uart**(const *esp_console_dev_uart_config_t* *dev_config, const *esp_console_repl_config_t* *repl_config, *esp_console_repl_t* **ret_repl)

Establish a console REPL environment over UART driver.

Attention This function is meant to be used in the examples to make the code more compact. Applications which use console functionality should be based on the underlying linenoise and esp_console functions.

Note: This is an all-in-one function to establish the environment needed for REPL, includes:

- Install the UART driver on the console UART (8n1, 115200, REF_TICK clock source)
- Configures the stdin/stdout to go through the UART driver
- Initializes linenoise
- Spawn new thread to run REPL in the background

Parameters

- **dev_config** –**[in]** UART device configuration
- **repl_config** –**[in]** REPL configuration
- **ret_repl** –**[out]** return REPL handle after initialization succeed, return NULL otherwise

Returns

- ESP_OK on success
- ESP_FAIL Parameter error

esp_err_t **esp_console_start_repl**(*esp_console_repl_t* *repl)

Start REPL environment.

Note: Once the REPL gets started, it won't be stopped until the user calls repl->del(repl) to destroy the REPL environment.

Parameters **repl** –**[in]** REPL handle returned from esp_console_new_repl_xxx

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE, if repl has started already

Structures

struct **esp_console_config_t**

Parameters for console initialization.

Public Members

size_t **max_cmdline_length**

length of command line buffer, in bytes

size_t **max_cmdline_args**

maximum number of command line arguments to parse

int **hint_color**

ASCII color code of hint text.

int **hint_bold**

Set to 1 to print hint text in bold.

struct **esp_console_repl_config_t**

Parameters for console REPL (Read Eval Print Loop)

Public Members

uint32_t **max_history_len**

maximum length for the history

const char ***history_save_path**

file path used to save history commands, set to NULL won't save to file system

uint32_t **task_stack_size**

repl task stack size

uint32_t **task_priority**

repl task priority

const char ***prompt**

prompt (NULL represents default: "esp> ")

size_t **max_cmdline_length**

maximum length of a command line. If 0, default value will be used

struct **esp_console_dev_uart_config_t**

Parameters for console device: UART.

Public Members

int **channel**

UART channel number (count from zero)

int **baud_rate**

Comunication baud rate.

int **tx_gpio_num**

GPIO number for TX path, -1 means using default one.

int **rx_gpio_num**

GPIO number for RX path, -1 means using default one.

struct **esp_console_cmd_t**

Console command description.

Public Members

const char ***command**

Command name. Must not be NULL, must not contain spaces. The pointer must be valid until the call to esp_console_deinit.

const char ***help**

Help text for the command, shown by help command. If set, the pointer must be valid until the call to esp_console_deinit. If not set, the command will not be listed in 'help' output.

const char ***hint**

Hint text, usually lists possible arguments. If set to NULL, and 'argtable' field is non-NULL, hint will be generated automatically

esp_console_cmd_func_t **func**

Pointer to a function which implements the command.

void ***argtable**

Array or structure of pointers to arg_xxx structures, may be NULL. Used to generate hint text if 'hint' is set to NULL. Array/structure which this field points to must end with an arg_end. Only used for the duration of esp_console_cmd_register call.

struct **esp_console_repl_s**

Console REPL base structure.

Public Members

esp_err_t (***del**)(*esp_console_repl_t* *repl)

Delete console REPL environment.

Param repl **[in]** REPL handle returned from esp_console_new_repl_xxx

Return

- ESP_OK on success
- ESP_FAIL on errors

Macros

ESP_CONSOLE_CONFIG_DEFAULT()

Default console configuration value.

ESP_CONSOLE_REPL_CONFIG_DEFAULT()

Default console repl configuration value.

ESP_CONSOLE_DEV_UART_CONFIG_DEFAULT()

Type Definitions

typedef struct *linenoiseCompletions* **linenoiseCompletions**

typedef int (***esp_console_cmd_func_t**)(int argc, char **argv)

Console command main function.

Param argc number of arguments
Param argv array with argc entries, each pointing to a zero-terminated string argument
Return console command return code, 0 indicates "success"

typedef struct *esp_console_repl_s* **esp_console_repl_t**

Type defined for console REPL.

2.10.6 eFuse Manager

Introduction

The eFuse Manager library is designed to structure access to eFuse bits and make using these easy. This library operates eFuse bits by a structure name which is assigned in eFuse table. This sections introduces some concepts used by eFuse Manager.

Hardware description

The ESP32 has a number of eFuses which can store system and user parameters. Each eFuse is a one-bit field which can be programmed to 1 after which it cannot be reverted back to 0. Some of system parameters are using these eFuse bits directly by hardware modules and have special place (for example EFUSE_BLK0).

For more details, see *ESP32 Technical Reference Manual* > *eFuse Controller (eFuse)* [PDF]. Some eFuse bits are available for user applications.

ESP32 has 4 eFuse blocks each of the size of 256 bits (not all bits are available):

- EFUSE_BLK0 is used entirely for system purposes;
- EFUSE_BLK1 is used for flash encrypt key. If not using that Flash Encryption feature, they can be used for another purpose;
- EFUSE_BLK2 is used for security boot key. If not using that Secure Boot feature, they can be used for another purpose;
- EFUSE_BLK3 can be partially reserved for the custom MAC address, or used entirely for user application. Note that some bits are already used in IDF.

Each block is divided into 8 32-bits registers.

eFuse Manager component

The component has API functions for reading and writing fields. Access to the fields is carried out through the structures that describe the location of the eFuse bits in the blocks. The component provides the ability to form fields of any length and from any number of individual bits. The description of the fields is made in a CSV file in a table form. To generate from a tabular form (CSV file) in the C-source uses the tool *efuse_table_gen.py*. The tool checks the CSV file for uniqueness of field names and bit intersection, in case of using a *custom* file from the user's project directory, the utility will check with the *common* CSV file.

CSV files:

- common (*esp_efuse_table.csv*) - contains eFuse fields which are used inside the IDF. C-source generation should be done manually when changing this file (run command `idf.py efuse-common-table`). Note that changes in this file can lead to incorrect operation.
- custom - (optional and can be enabled by *CONFIG_EFUSE_CUSTOM_TABLE*) contains eFuse fields that are used by the user in their application. C-source generation should be done manually when changing this file and running `idf.py efuse-custom-table`.

Description CSV file

The CSV file contains a description of the eFuse fields. In the simple case, one field has one line of description. Table header:

```
# field_name,  efuse_block(EFUSE_BLK0..EFUSE_BLK3), bit_start(0..255),    bit_
↪count(1..256),         comment
```

Individual params in CSV file the following meanings:

field_name Name of field. The prefix *ESP_EFUSE_* will be added to the name, and this field name will be available in the code. This name will be used to access the fields. The name must be unique for all fields. If the line has an empty name, then this line is combined with the previous field. This allows you to set an arbitrary order of bits in the field, and expand the field as well (see `MAC_FACTORY` field in the common table). The field_name supports structured format using . to show that the field belongs to another field (see `WR_DIS` and `RD_DIS` in the common table).

efuse_block Block number. It determines where the eFuse bits will be placed for this field. Available EFUSE_BLK0..EFUSE_BLK3.

bit_start Start bit number (0..255). The bit_start field can be omitted. In this case, it will be set to bit_start + bit_count from the previous record, if it has the same efuse_block. Otherwise (if efuse_block is different, or this is the first entry), an error will be generated.

bit_count The number of bits to use in this field (1..-). This parameter can not be omitted. This field also may be `MAX_BLK_LEN` in this case, the field length will have the maximum block length, taking into account the coding scheme (applicable for `ESP_EFUSE_SECURE_BOOT_KEY` and `ESP_EFUSE_ENCRYPT_FLASH_KEY` fields). The value `MAX_BLK_LEN` depends on *CONFIG_EFUSE_CODE_SCHEME_SELECTOR*, will be replaced with "None" - 256, "3/4" - 192, "REPEAT" - 128.

comment This param is using for comment field, it also move to C-header file. The comment field can be omitted.

If a non-sequential bit order is required to describe a field, then the field description in the following lines should be continued without specifying a name, this will indicate that it belongs to one field. For example two fields MAC_FACTORY and MAC_FACTORY_CRC:

```
# Factory MAC address #
#######################
MAC_FACTORY,            EFUSE_BLK0,     72,     8,    Factory MAC addr [0]
,                       EFUSE_BLK0,     64,     8,    Factory MAC addr [1]
,                       EFUSE_BLK0,     56,     8,    Factory MAC addr [2]
,                       EFUSE_BLK0,     48,     8,    Factory MAC addr [3]
,                       EFUSE_BLK0,     40,     8,    Factory MAC addr [4]
,                       EFUSE_BLK0,     32,     8,    Factory MAC addr [5]
MAC_FACTORY_CRC,        EFUSE_BLK0,     80,     8,    CRC8 for factory MAC address
```

This field will available in code as ESP_EFUSE_MAC_FACTORY and ESP_EFUSE_MAC_FACTORY_CRC.

Structured efuse fields

```
WR_DIS,                         EFUSE_BLK0,    0,     32,     Write protection
WR_DIS.RD_DIS,                  EFUSE_BLK0,    0,     1,      Write protection for␣
↪RD_DIS
```

(continues on next page)

(continued from previous page)

```
WR_DIS.FIELD_1,                    EFUSE_BLK0,    1,     1,       Write protection for␣
↪FIELD_1
WR_DIS.FIELD_2,                    EFUSE_BLK0,    2,     4,       Write protection for␣
↪FIELD_2 (includes B1 and B2)
WR_DIS.FIELD_2.B1,                 EFUSE_BLK0,    2,     2,       Write protection for␣
↪FIELD_2.B1
WR_DIS.FIELD_2.B2,                 EFUSE_BLK0,    4,     2,       Write protection for␣
↪FIELD_2.B2
WR_DIS.FIELD_3,                    EFUSE_BLK0,    5,     1,       Write protection for␣
↪FIELD_3
WR_DIS.FIELD_3.ALIAS,              EFUSE_BLK0,    5,     1,       Write protection for␣
↪FIELD_3 (just a alias for WR_DIS.FIELD_3)
WR_DIS.FIELD_4,                    EFUSE_BLK0,    7,     1,       Write protection for␣
↪FIELD_4
```

The structured eFuse field looks like `WR_DIS.RD_DIS` where the dot points that this field belongs to the parent field - `WR_DIS` and can not be out of the parent's range.

It is possible to use some levels of structured fields as WR_DIS.FIELD_2.B1 and B2. These fields should not be crossed each other and should be in the range of two fields: `WR_DIS` and `WR_DIS.FIELD_2`.

It is possible to create aliases for fields with the same range, see `WR_DIS.FIELD_3` and `WR_DIS.FIELD_3.ALIAS`.

The IDF names for structured efuse fields should be unique. The `efuse_table_gen` tool will generate the final names where the dot will be replaced by _. The names for using in IDF are ESP_EFUSE_WR_DIS, ESP_EFUSE_WR_DIS_RD_DIS, ESP_EFUSE_WR_DIS_FIELD_2_B1, etc.

The `efuse_table_gen` tool checks that the fields do not overlap each other and must be within the range of a field if there is a violation, then throws the following error:

```
Field at USER_DATA, EFUSE_BLK3, 0, 256  intersected with  SERIAL_NUMBER, EFUSE_
↪BLK3, 0, 32
```

Solution: Describe `SERIAL_NUMBER` to be included in `USER_DATA`. (`USER_DATA.SERIAL_NUMBER`).

```
Field at FEILD, EFUSE_BLK3, 0, 50  out of range  FEILD.MAJOR_NUMBER, EFUSE_BLK3,␣
↪60, 32
```

Solution: Change `bit_start` for `FIELD.MAJOR_NUMBER` from 60 to 0, so `MAJOR_NUMBER` is in the `FEILD` range.

efuse_table_gen.py tool

The tool is designed to generate C-source files from CSV file and validate fields. First of all, the check is carried out on the uniqueness of the names and overlaps of the field bits. If an additional *custom* file is used, it will be checked with the existing *common* file (esp_efuse_table.csv). In case of errors, a message will be displayed and the string that caused the error. C-source files contain structures of type *esp_efuse_desc_t*.

To generate a *common* files, use the following command `idf.py efuse-common-table` or:

```
cd $IDF_PATH/components/efuse/
./efuse_table_gen.py --idf_target esp32 esp32/esp_efuse_table.csv
```

After generation in the folder $IDF_PATH/components/efuse/*esp32* create:

- *esp_efuse_table.c* file.
- In *include* folder *esp_efuse_table.c* file.

To generate a *custom* files, use the following command `idf.py efuse-custom-table` or:

```
cd $IDF_PATH/components/efuse/
./efuse_table_gen.py --idf_target esp32 esp32/esp_efuse_table.csv PROJECT_PATH/
↪main/esp_efuse_custom_table.csv
```

After generation in the folder PROJECT_PATH/main create:

- *esp_efuse_custom_table.c* file.
- In *include* folder *esp_efuse_custom_table.c* file.

To use the generated fields, you need to include two files:

```
#include "esp_efuse.h"
#include "esp_efuse_table.h" // or "esp_efuse_custom_table.h"
```

Supported coding scheme

eFuse have three coding schemes:

- `None` (value 0).
- `3/4` (value 1).
- `Repeat` (value 2).

The coding scheme affects only EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3 blocks. EUSE_BLK0 block always has a coding scheme `None`. Coding changes the number of bits that can be written into a block, the block length is constant 256, some of these bits are used for encoding and not avaliable for the user.

When using a coding scheme, the length of the payload that can be written is limited (for more details `20.3.1.3 System Parameter coding_scheme`):

- None 256 bits.
- 3/4 192 bits.
- Repeat 128 bits.

You can find out the coding scheme of your chip:

- run a `espefuse.py -p PORT summary` command.
- from `esptool` utility logs (during flashing).
- calling the function in the code *`esp_efuse_get_coding_scheme()`* for the EFUSE_BLK3 block.

eFuse tables must always comply with the coding scheme in the chip. There is an *CONFIG_EFUSE_CODE_SCHEME_SELECTOR* option to select the coding type for tables in a Kconfig. When generating source files, if your tables do not follow the coding scheme, an error message will be displayed. Adjust the length or offset fields. If your program was compiled with `None` encoding and `3/4` is used in the chip, then the `ESP_ERR_CODING` error may occur when calling the eFuse API (the field is outside the block boundaries). If the field matches the new block boundaries, then the API will work without errors.

Also, 3/4 coding scheme imposes restrictions on writing bits belonging to one coding unit. The whole block with a length of 256 bits is divided into 4 coding units, and in each coding unit there are 6 bytes of useful data and 2 service bytes. These 2 service bytes contain the checksum of the previous 6 data bytes.

It turns out that only one field can be written into one coding unit. Repeated rewriting in one coding unit is prohibited. But if the record was made in advance or through a *`esp_efuse_write_block()`* function, then reading the fields belonging to one coding unit is possible.

In case `3/4` coding scheme, the writing process is divided into the coding units and we can not use the usual mode of writing some fields. We can prepare all the data for writing and burn it in one time. You can also use this mode for `None` coding scheme but it is not necessary. It is important for `3/4` coding scheme. `The batch writing mode` blocks `esp_efuse_read_...` operations.

After changing the coding scheme, run `efuse_common_table` and `efuse_custom_table` commands to check the tables of the new coding scheme.

To write some fields into one block, or different blocks in one time, you need to use `the batch writing mode`. Firstly set this mode through *`esp_efuse_batch_write_begin()`* function then write

some fields as usual using the `esp_efuse_write_...` functions. At the end to burn them, call the *`esp_efuse_batch_write_commit()`* function. It burns prepared data to the eFuse blocks and disables the `batch recording mode`.

Note: If there is already pre-written data in the eFuse block using the `3/4 or Repeat` encoding scheme, then it is not possible to write anything extra (even if the required bits are empty) without breaking the previous encoding data. This encoding data will be overwritten with new encoding data and completely destroyed (however, the payload eFuses are not damaged). It can be related to: CUSTOM_MAC, SPI_PAD_CONFIG_HD, SPI_PAD_CONFIG_CS, etc. Please contact Espressif to order the required pre-burnt eFuses.

FOR TESTING ONLY (NOT RECOMMENDED): You can ignore or suppress errors that violate encoding scheme data in order to burn the necessary bits in the eFuse block.

---

eFuse API

Access to the fields is via a pointer to the description structure. API functions have some basic operation:

- *`esp_efuse_read_field_blob()`* - returns an array of read eFuse bits.
- *`esp_efuse_read_field_cnt()`* - returns the number of bits programmed as "1".
- *`esp_efuse_write_field_blob()`* - writes an array.
- *`esp_efuse_write_field_cnt()`* - writes a required count of bits as "1".
- *`esp_efuse_get_field_size()`* - returns the number of bits by the field name.
- *`esp_efuse_read_reg()`* - returns value of eFuse register.
- *`esp_efuse_write_reg()`* - writes value to eFuse register.
- *`esp_efuse_get_coding_scheme()`* - returns eFuse coding scheme for blocks.
- *`esp_efuse_read_block()`* - reads key to eFuse block starting at the offset and the required size.
- *`esp_efuse_write_block()`* - writes key to eFuse block starting at the offset and the required size.
- *`esp_efuse_batch_write_begin()`* - set the batch mode of writing fields.
- *`esp_efuse_batch_write_commit()`* - writes all prepared data for batch writing mode and reset the batch writing mode.
- *`esp_efuse_batch_write_cancel()`* - reset the batch writing mode and prepared data.
- *`esp_efuse_get_key_dis_read()`* - Returns a read protection for the key block.
- *`esp_efuse_set_key_dis_read()`* - Sets a read protection for the key block.
- *`esp_efuse_get_key_dis_write()`* - Returns a write protection for the key block.
- *`esp_efuse_set_key_dis_write()`* - Sets a write protection for the key block.
- *`esp_efuse_get_key_purpose()`* - Returns the current purpose set for an eFuse key block.
- *`esp_efuse_write_key()`* - Programs a block of key data to an eFuse block
- *`esp_efuse_write_keys()`* - Programs keys to unused eFuse blocks
- *`esp_efuse_find_purpose()`* - Finds a key block with the particular purpose set.
- *`esp_efuse_get_keypurpose_dis_write()`* - Returns a write protection of the key purpose field for an eFuse key block (for esp32 always true).
- *`esp_efuse_key_block_unused()`* - Returns true if the key block is unused, false otherwise.

For frequently used fields, special functions are made, like this *`esp_efuse_get_pkg_ver()`*.

How to add a new field

1. Find a free bits for field. Show *esp_efuse_table.csv* file or run `idf.py show-efuse-table` or the next command:

```
$ ./efuse_table_gen.py esp32/esp_efuse_table.csv --info

Parsing efuse CSV input file $IDF_PATH/components/efuse/esp32/esp_efuse_table.csv .
↪..
Verifying efuse table...
Max number of bits in BLK 192
```

(continues on next page)

(continued from previous page)

```
Sorted efuse table:
#       field_name                      efuse_block     bit_start       bit_count
1       WR_DIS_EFUSE_RD_DISABLE         EFUSE_BLK0          0               1
2       WR_DIS_FLASH_CRYPT_CNT          EFUSE_BLK0          2               1
3       WR_DIS_BLK1                     EFUSE_BLK0          7               1
4       WR_DIS_BLK2                     EFUSE_BLK0          8               1
5       WR_DIS_BLK3                     EFUSE_BLK0          9               1
6       RD_DIS_BLK1                     EFUSE_BLK0          16              1
7       RD_DIS_BLK2                     EFUSE_BLK0          17              1
8       RD_DIS_BLK3                     EFUSE_BLK0          18              1
9       FLASH_CRYPT_CNT                 EFUSE_BLK0          20              7
10      UART_DOWNLOAD_DIS               EFUSE_BLK0          27              1
11      MAC_FACTORY                     EFUSE_BLK0          32              8
12      MAC_FACTORY                     EFUSE_BLK0          40              8
13      MAC_FACTORY                     EFUSE_BLK0          48              8
14      MAC_FACTORY                     EFUSE_BLK0          56              8
15      MAC_FACTORY                     EFUSE_BLK0          64              8
16      MAC_FACTORY                     EFUSE_BLK0          72              8
17      MAC_FACTORY_CRC                 EFUSE_BLK0          80              8
18      CHIP_VER_DIS_APP_CPU            EFUSE_BLK0          96              1
19      CHIP_VER_DIS_BT                 EFUSE_BLK0          97              1
20      CHIP_VER_PKG                    EFUSE_BLK0          98              1
21      CHIP_VER_PKG                    EFUSE_BLK0          105             3
22      CHIP_CPU_FREQ_LOW               EFUSE_BLK0          108             1
23      CHIP_CPU_FREQ_RATED             EFUSE_BLK0          109             1
24      CHIP_VER_REV1                   EFUSE_BLK0          111             1
25      ADC_VREF_AND_SDIO_DREF          EFUSE_BLK0          136             6
26      XPD_SDIO_REG                    EFUSE_BLK0          142             1
27      SDIO_TIEH                       EFUSE_BLK0          143             1
28      SDIO_FORCE                      EFUSE_BLK0          144             1
29      CHIP_VER_REV2                   EFUSE_BLK0          180             1
30      ENCRYPT_CONFIG                  EFUSE_BLK0          188             4
31      CONSOLE_DEBUG_DISABLE           EFUSE_BLK0          194             1
32      ABS_DONE_0                      EFUSE_BLK0          196             1
33      ABS_DONE_1                      EFUSE_BLK0          197             1
34      DISABLE_JTAG                    EFUSE_BLK0          198             1
35      DISABLE_DL_ENCRYPT              EFUSE_BLK0          199             1
36      DISABLE_DL_DECRYPT              EFUSE_BLK0          200             1
37      DISABLE_DL_CACHE                EFUSE_BLK0          201             1
38      ENCRYPT_FLASH_KEY               EFUSE_BLK1          0               192
39      SECURE_BOOT_KEY                 EFUSE_BLK2          0               192
40      MAC_CUSTOM_CRC                  EFUSE_BLK3          0               8
41      MAC_CUSTOM                      EFUSE_BLK3          8               48
42      ADC1_TP_LOW                     EFUSE_BLK3          96              7
43      ADC1_TP_HIGH                    EFUSE_BLK3          103             9
44      ADC2_TP_LOW                     EFUSE_BLK3          112             7
45      ADC2_TP_HIGH                    EFUSE_BLK3          119             9
46      SECURE_VERSION                  EFUSE_BLK3          128             32
47      MAC_CUSTOM_VER                  EFUSE_BLK3          184             8

Used bits in efuse table:
EFUSE_BLK0
[0 0] [2 2] [7 9] [16 18] [20 27] [32 87] [96 98] [105 109] [111 111] [136 144]␣
↪[180 180] [188 191] [194 194] [196 201]

EFUSE_BLK1
[0 191]

EFUSE_BLK2
[0 191]
```

(continues on next page)

(continued from previous page)

```
EFUSE_BLK3
[0 55] [96 159] [184 191]

Note: Not printed ranges are free for using. (bits in EFUSE_BLK0 are reserved for␣
↪Espressif)
```

The number of bits not included in square brackets is free (some bits are reserved for Espressif). All fields are checked for overlapping.

To add fields to an existing field, use the *Structured efuse fields* technique. For example, adding the fields: SERIAL_NUMBER, MODEL_NUMBER and HARDWARE REV to an existing `USER_DATA` field. Use `.` (dot) to show an attachment in a field.

```
USER_DATA.SERIAL_NUMBER,                    EFUSE_BLK3,     0,  32,
USER_DATA.MODEL_NUMBER,                     EFUSE_BLK3,     32, 10,
USER_DATA.HARDWARE_REV,                     EFUSE_BLK3,     42, 10,
```

2. Fill a line for field: field_name, efuse_block, bit_start, bit_count, comment.
3. Run a `show_efuse_table` command to check eFuse table. To generate source files run `efuse_common_table` or `efuse_custom_table` command.

You may get errors such as `intersects with` or `out of range`. Please see how to solve them in the *Structured efuse fields* article.

Bit Order

The eFuses bit order is little endian (see the example below), it means that eFuse bits are read and written from LSB to MSB:

```
$ espefuse.py dump

USER_DATA       (BLOCK3          ) [3 ] read_regs: 03020100 07060504 0B0A0908␣
↪0F0E0D0C 13121111 17161514 1B1A1918 1F1E1D1C
BLOCK4          (BLOCK4          ) [4 ] read_regs: 03020100 07060504 0B0A0908␣
↪0F0E0D0C 13121111 17161514 1B1A1918 1F1E1D1C

where is the register representation:

EFUSE_RD_USR_DATA0_REG = 0x03020100
EFUSE_RD_USR_DATA1_REG = 0x07060504
EFUSE_RD_USR_DATA2_REG = 0x0B0A0908
EFUSE_RD_USR_DATA3_REG = 0x0F0E0D0C
EFUSE_RD_USR_DATA4_REG = 0x13121111
EFUSE_RD_USR_DATA5_REG = 0x17161514
EFUSE_RD_USR_DATA6_REG = 0x1B1A1918
EFUSE_RD_USR_DATA7_REG = 0x1F1E1D1C

where is the byte representation:

byte[0] = 0x00, byte[1] = 0x01, ... byte[3] = 0x03, byte[4] = 0x04, ..., byte[31]␣
↪= 0x1F
```

For example, csv file describes the `USER_DATA` field, which occupies all 256 bits (a whole block).

```
USER_DATA,          EFUSE_BLK3,     0,  256,     User data
USER_DATA.FIELD1,   EFUSE_BLK3,     16,  16,     Field1

ID,                 EFUSE_BLK4,     8,  3,      ID bit[0..2]
,                   EFUSE_BLK4,     16, 2,      ID bit[3..4]
,                   EFUSE_BLK4,     32, 3,      ID bit[5..7]
```

Thus, reading the eFuse `USER_DATA` block written as above gives the following results:

```
uint8_t buf[32] = { 0 };
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &buf, sizeof(buf) * 8);
// buf[0] = 0x00, buf[1] = 0x01, ... buf[31] = 0x1F

uint32_t field1 = 0;
size_t field1_size = ESP_EFUSE_USER_DATA[0]->bit_count; // can be used for this␣
↪case because it only consists of one entry
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &field1, field1_size);
// field1 = 0x0302

uint32_t field1_1 = 0;
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &field1_1, 2); // reads only first␣
↪2 bits
// field1 = 0x0002

uint8_t id = 0;
size_t id_size = esp_efuse_get_field_size(ESP_EFUSE_ID); // returns 6
// size_t id_size = ESP_EFUSE_USER_DATA[0]->bit_count; // can NOT be used because␣
↪it consists of 3 entries. It returns 3 not 6.
esp_efuse_read_field_blob(ESP_EFUSE_ID, &id, id_size);
// id = 0x91
// b'100 10  001
//   [3] [2] [3]

uint8_t id_1 = 0;
esp_efuse_read_field_blob(ESP_EFUSE_ID, &id_1, 3);
// id = 0x01
// b'001
```

Debug eFuse & Unit tests

Virtual eFuses The Kconfig option *CONFIG_EFUSE_VIRTUAL* will virtualize eFuse values inside the eFuse Manager, so writes are emulated and no eFuse values are permanently changed. This can be useful for debugging app and unit tests. During startup, the eFuses are copied to RAM. All eFuse operations (read and write) are performed with RAM instead of the real eFuse registers.

In addition to the *CONFIG_EFUSE_VIRTUAL* option there is *CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH* option that adds a feature to keep eFuses in flash memory. To use this mode the partition_table should have the *efuse* partition. partition.csv: "`efuse_em, data, efuse, , 0x2000,`". During startup, the eFuses are copied from flash or, in case if flash is empty, from real eFuse to RAM and then update flash. This option allows keeping eFuses after reboots (possible to test secure_boot and flash_encryption features with this option).

Flash Encryption Testing Flash Encryption (FE) is a hardware feature that requires the physical burning of eFuses: key and FLASH_CRYPT_CNT. If FE is not actually enabled then enabling the *CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH* option just gives testing possibilities and does not encrypt anything in the flash, even though the logs say encryption happens. The `bootloader_flash_write()` is adapted for this purpose. But if FE is already enabled on the chip and you run an application or bootloader created with the *CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH* option then the flash encryption/decryption operations will work properly (data are encrypted as it is written into an encrypted flash partition and decrypted when they are read from an encrypted partition).

espefuse.py esptool includes a useful tool for reading/writing ESP32 eFuse bits - espefuse.py.

```
espefuse.py -p PORT summary

Connecting........__
```

(continues on next page)

(continued from previous page)

```
Detecting chip type... ESP32
espefuse.py v3.1-dev
EFUSE_NAME (Block)                      Description  = [Meaningful Value]␣
↪[Readable/Writeable] (Hex Value)
----------------------------------------------------------------------------␣
↪-----
Calibration fuses:
BLK3_PART_RESERVE (BLOCK0):             BLOCK3 partially served for ADC␣
↪calibration data    = True R/W (0b1)
ADC_VREF (BLOCK0):                      Voltage reference calibration          ␣
↪          = 1114 R/W (0b00010)
ADC1_TP_LOW (BLOCK3):                   ADC1 150mV reading                     ␣
↪          = 346 R/W (0b0010001)
ADC1_TP_HIGH (BLOCK3):                  ADC1 850mV reading                     ␣
↪          = 3285 R/W (0b000000101)
ADC2_TP_LOW (BLOCK3):                   ADC2 150mV reading                     ␣
↪          = 449 R/W (0b0000111)
ADC2_TP_HIGH (BLOCK3):                  ADC2 850mV reading                     ␣
↪          = 3362 R/W (0b111110101)

Config fuses:
XPD_SDIO_FORCE (BLOCK0):                Ignore MTDI pin (GPIO12) for VDD_SDIO on␣
↪reset     = False R/W (0b0)
XPD_SDIO_REG (BLOCK0):                  If XPD_SDIO_FORCE, enable VDD_SDIO reg on␣
↪reset    = False R/W (0b0)
XPD_SDIO_TIEH (BLOCK0):                 If XPD_SDIO_FORCE & XPD_SDIO_REG       ␣
↪          = 1.8V R/W (0b0)
CLK8M_FREQ (BLOCK0):                    8MHz clock freq override               ␣
↪          = 53 R/W (0x35)
SPI_PAD_CONFIG_CLK (BLOCK0):            Override SD_CLK pad (GPIO6/SPICLK)     ␣
↪          = 0 R/W (0b00000)
SPI_PAD_CONFIG_Q (BLOCK0):              Override SD_DATA_0 pad (GPIO7/SPIQ)    ␣
↪          = 0 R/W (0b00000)
SPI_PAD_CONFIG_D (BLOCK0):              Override SD_DATA_1 pad (GPIO8/SPID)    ␣
↪          = 0 R/W (0b00000)
SPI_PAD_CONFIG_HD (BLOCK0):             Override SD_DATA_2 pad (GPIO9/SPIHD)   ␣
↪          = 0 R/W (0b00000)
SPI_PAD_CONFIG_CS0 (BLOCK0):            Override SD_CMD pad (GPIO11/SPICS0)    ␣
↪          = 0 R/W (0b00000)
DISABLE_SDIO_HOST (BLOCK0):             Disable SDIO host                      ␣
↪          = False R/W (0b0)

Efuse fuses:
WR_DIS (BLOCK0):                        Efuse write disable mask               ␣
↪          = 0 R/W (0x0000)
RD_DIS (BLOCK0):                        Efuse read disable mask                ␣
↪          = 0 R/W (0x0)
CODING_SCHEME (BLOCK0):                 Efuse variable block length scheme
= 3/4 (BLK1-3 len=192 bits) R/W (0b01)
KEY_STATUS (BLOCK0):                    Usage of efuse block 3 (reserved)      ␣
↪          = False R/W (0b0)

Identity fuses:
MAC (BLOCK0):                           Factory MAC Address
= 84:0d:8e:18:8e:44 (CRC 0xad OK) R/W
MAC_CRC (BLOCK0):                       CRC8 for factory MAC address           ␣
↪          = 173 R/W (0xad)
CHIP_VER_REV1 (BLOCK0):                 Silicon Revision 1                     ␣
↪          = True R/W (0b1)
CHIP_VER_REV2 (BLOCK0):                 Silicon Revision 2                     ␣
↪          = False R/W (0b0)
```

(continues on next page)

(continued from previous page)

```
CHIP_VERSION (BLOCK0):                    Reserved for future chip versions
↪          = 2 R/W (0b10)
CHIP_PACKAGE (BLOCK0):                    Chip package identifier
↪          = 0 R/W (0b000)
MAC_VERSION (BLOCK3):                     Version of the MAC field
↪          = 0 R/W (0x00)

Security fuses:
FLASH_CRYPT_CNT (BLOCK0):                 Flash encryption mode counter
↪          = 0 R/W (0b0000000)
UART_DOWNLOAD_DIS (BLOCK0):               Disable UART download mode (ESP32 rev3
↪only)        = False R/W (0b0)
FLASH_CRYPT_CONFIG (BLOCK0):              Flash encryption config (key tweak bits)
↪          = 0 R/W (0x0)
CONSOLE_DEBUG_DISABLE (BLOCK0):           Disable ROM BASIC interpreter fallback
↪          = True R/W (0b1)
ABS_DONE_0 (BLOCK0):                      Secure boot V1 is enabled for bootloader
↪image      = False R/W (0b0)
ABS_DONE_1 (BLOCK0):                      Secure boot V2 is enabled for bootloader
↪image      = False R/W (0b0)
JTAG_DISABLE (BLOCK0):                    Disable JTAG
↪          = False R/W (0b0)
DISABLE_DL_ENCRYPT (BLOCK0):              Disable flash encryption in UART
↪bootloader         = False R/W (0b0)
DISABLE_DL_DECRYPT (BLOCK0):              Disable flash decryption in UART
↪bootloader         = False R/W (0b0)
DISABLE_DL_CACHE (BLOCK0):                Disable flash cache in UART bootloader
↪          = False R/W (0b0)
BLOCK1 (BLOCK1):                          Flash encryption key
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
BLOCK2 (BLOCK2):                          Secure boot key
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
BLOCK3 (BLOCK3):                          Variable Block 3
= 00 00 00 00 00 00 00 00 00 00 00 91 02 87 fa 00 00 00 00 00 00 00 00 00 R/W

Flash voltage (VDD_SDIO) determined by GPIO12 on reset (High for 1.8V, Low/NC for
↪3.3V).
```

To get a dump for all eFuse registers.

```
espefuse.py -p PORT dump

Connecting........_
Detecting chip type... ESP32
BLOCK0          (                ) [0 ] read_regs: 00000000 8e188e44 00ad840d
↪0000e000 00000235 00000000 00000005
BLOCK1          (flash_encryption) [1 ] read_regs: 00000000 00000000 00000000
↪00000000 00000000 00000000
BLOCK2          (secure_boot_v1 s) [2 ] read_regs: 00000000 00000000 00000000
↪00000000 00000000 00000000
BLOCK3          (                ) [3 ] read_regs: 00000000 00000000 00000000
↪fa870291 00000000 00000000
espefuse.py v3.1-dev
```

Header File

- components/efuse/esp32/include/esp_efuse_chip.h

Enumerations

enum **esp_efuse_block_t**

Type of eFuse blocks for ESP32.

Values:

enumerator **EFUSE_BLK0**

Number of eFuse block. Reserved.

enumerator **EFUSE_BLK1**

Number of eFuse block. Used for Flash Encryption. If not using that Flash Encryption feature, they can be used for another purpose.

enumerator **EFUSE_BLK_KEY0**

Number of eFuse block. Used for Flash Encryption. If not using that Flash Encryption feature, they can be used for another purpose.

enumerator **EFUSE_BLK_ENCRYPT_FLASH**

Number of eFuse block. Used for Flash Encryption. If not using that Flash Encryption feature, they can be used for another purpose.

enumerator **EFUSE_BLK2**

Number of eFuse block. Used for Secure Boot. If not using that Secure Boot feature, they can be used for another purpose.

enumerator **EFUSE_BLK_KEY1**

Number of eFuse block. Used for Secure Boot. If not using that Secure Boot feature, they can be used for another purpose.

enumerator **EFUSE_BLK_SECURE_BOOT**

Number of eFuse block. Used for Secure Boot. If not using that Secure Boot feature, they can be used for another purpose.

enumerator **EFUSE_BLK3**

Number of eFuse block. Uses for the purpose of the user.

enumerator **EFUSE_BLK_KEY2**

Number of eFuse block. Uses for the purpose of the user.

enumerator **EFUSE_BLK_KEY_MAX**

enumerator **EFUSE_BLK_MAX**

enum **esp_efuse_coding_scheme_t**

Type of coding scheme.

Values:

enumerator **EFUSE_CODING_SCHEME_NONE**

None

enumerator **EFUSE_CODING_SCHEME_3_4**

3/4 coding

enumerator **EFUSE_CODING_SCHEME_REPEAT**

Repeat coding

enum **esp_efuse_purpose_t**

Type of key purpose (virtual because ESP32 has only fixed purposes for blocks)

Values:

enumerator **ESP_EFUSE_KEY_PURPOSE_USER**

BLOCK3

enumerator **ESP_EFUSE_KEY_PURPOSE_SYSTEM**

BLOCK0

enumerator **ESP_EFUSE_KEY_PURPOSE_FLASH_ENCRYPTION**

BLOCK1

enumerator **ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_V2**

BLOCK2

enumerator **ESP_EFUSE_KEY_PURPOSE_MAX**

MAX PURPOSE

Header File

- components/efuse/include/esp_efuse.h

Functions

esp_err_t **esp_efuse_read_field_blob**(const *esp_efuse_desc_t* *field[], void *dst, size_t dst_size_bits)

Reads bits from EFUSE field and writes it into an array.

The number of read bits will be limited to the minimum value from the description of the bits in "field" structure or "dst_size_bits" required size. Use "esp_efuse_get_field_size()" function to determine the length of the field.

Note: Please note that reading in the batch mode does not show uncommitted changes.

Parameters

- **field** –**[in]** A pointer to the structure describing the fields of efuse.
- **dst** –**[out]** A pointer to array that will contain the result of reading.
- **dst_size_bits** –**[in]** The number of bits required to read. If the requested number of bits is greater than the field, the number will be limited to the field size.

Returns

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

bool **esp_efuse_read_field_bit**(const *esp_efuse_desc_t* *field[])

Read a single bit eFuse field as a boolean value.

Note: The value must exist and must be a single bit wide. If there is any possibility of an error in the provided arguments, call esp_efuse_read_field_blob() and check the returned value instead.

Note: If assertions are enabled and the parameter is invalid, execution will abort

Note: Please note that reading in the batch mode does not show uncommitted changes.

Parameters **`field`** –**[in]** A pointer to the structure describing the fields of efuse.

Returns

- true: The field parameter is valid and the bit is set.
- false: The bit is not set, or the parameter is invalid and assertions are disabled.

esp_err_t **`esp_efuse_read_field_cnt`**(const *esp_efuse_desc_t* *field[], size_t *out_cnt)

Reads bits from EFUSE field and returns number of bits programmed as "1".

If the bits are set not sequentially, they will still be counted.

Note: Please note that reading in the batch mode does not show uncommitted changes.

Parameters

- **`field`** –**[in]** A pointer to the structure describing the fields of efuse.
- **`out_cnt`** –**[out]** A pointer that will contain the number of programmed as "1" bits.

Returns

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

esp_err_t **`esp_efuse_write_field_blob`**(const *esp_efuse_desc_t* *field[], const void *src, size_t src_size_bits)

Writes array to EFUSE field.

The number of write bits will be limited to the minimum value from the description of the bits in "field" structure or "src_size_bits" required size. Use "esp_efuse_get_field_size()" function to determine the length of the field. After the function is completed, the writing registers are cleared.

Parameters

- **`field`** –**[in]** A pointer to the structure describing the fields of efuse.
- **`src`** –**[in]** A pointer to array that contains the data for writing.
- **`src_size_bits`** –**[in]** The number of bits required to write.

Returns

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

esp_err_t **`esp_efuse_write_field_cnt`**(const *esp_efuse_desc_t* *field[], size_t cnt)

Writes a required count of bits as "1" to EFUSE field.

If there are no free bits in the field to set the required number of bits to "1", ESP_ERR_EFUSE_CNT_IS_FULL error is returned, the field will not be partially recorded. After the function is completed, the writing registers are cleared.

Parameters

- **`field`** –**[in]** A pointer to the structure describing the fields of efuse.
- **`cnt`** –**[in]** Required number of programmed as "1" bits.

Returns

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

- ESP_ERR_EFUSE_CNT_IS_FULL: Not all requested cnt bits is set.

esp_err_t **esp_efuse_write_field_bit** (const *esp_efuse_desc_t* *field[])

Write a single bit eFuse field to 1.

For use with eFuse fields that are a single bit. This function will write the bit to value 1 if it is not already set, or does nothing if the bit is already set.

This is equivalent to calling esp_efuse_write_field_cnt() with the cnt parameter equal to 1, except that it will return ESP_OK if the field is already set to 1.

Parameters **field** –**[in]** Pointer to the structure describing the efuse field.
Returns
- ESP_OK: The operation was successfully completed, or the bit was already set to value 1.
- ESP_ERR_INVALID_ARG: Error in the passed arugments, including if the efuse field is not 1 bit wide.

esp_err_t **esp_efuse_set_write_protect** (*esp_efuse_block_t* blk)

Sets a write protection for the whole block.

After that, it is impossible to write to this block. The write protection does not apply to block 0.

Parameters **blk** –**[in]** Block number of eFuse. (EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3)
Returns
- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_CNT_IS_FULL: Not all requested cnt bits is set.
- ESP_ERR_NOT_SUPPORTED: The block does not support this command.

esp_err_t **esp_efuse_set_read_protect** (*esp_efuse_block_t* blk)

Sets a read protection for the whole block.

After that, it is impossible to read from this block. The read protection does not apply to block 0.

Parameters **blk** –**[in]** Block number of eFuse. (EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3)
Returns
- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_CNT_IS_FULL: Not all requested cnt bits is set.
- ESP_ERR_NOT_SUPPORTED: The block does not support this command.

int **esp_efuse_get_field_size** (const *esp_efuse_desc_t* *field[])

Returns the number of bits used by field.

Parameters **field** –**[in]** A pointer to the structure describing the fields of efuse.
Returns Returns the number of bits used by field.

uint32_t **esp_efuse_read_reg** (*esp_efuse_block_t* blk, unsigned int num_reg)

Returns value of efuse register.

This is a thread-safe implementation. Example: EFUSE_BLK2_RDATA3_REG where (blk=2, num_reg=3)

Note: Please note that reading in the batch mode does not show uncommitted changes.

Parameters
- **blk** –**[in]** Block number of eFuse.
- **num_reg** –**[in]** The register number in the block.

Returns Value of register

esp_err_t **esp_efuse_write_reg** (*esp_efuse_block_t* blk, unsigned int num_reg, uint32_t val)

Write value to efuse register.

Apply a coding scheme if necessary. This is a thread-safe implementation. Example: EFUSE_BLK3_WDATA0_REG where (blk=3, num_reg=0)

Parameters
- **blk** –**[in]** Block number of eFuse.
- **num_reg** –**[in]** The register number in the block.
- **val** –**[in]** Value to write.

Returns
- ESP_OK: The operation was successfully completed.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.

esp_efuse_coding_scheme_t **esp_efuse_get_coding_scheme** (*esp_efuse_block_t* blk)

Return efuse coding scheme for blocks.

Note: The coding scheme is applicable only to 1, 2 and 3 blocks. For 0 block, the coding scheme is always NONE.

Parameters **blk** –**[in]** Block number of eFuse.
Returns Return efuse coding scheme for blocks

esp_err_t **esp_efuse_read_block** (*esp_efuse_block_t* blk, void *dst_key, size_t offset_in_bits, size_t size_bits)

Read key to efuse block starting at the offset and the required size.

Note: Please note that reading in the batch mode does not show uncommitted changes.

Parameters
- **blk** –**[in]** Block number of eFuse.
- **dst_key** –**[in]** A pointer to array that will contain the result of reading.
- **offset_in_bits** –**[in]** Start bit in block.
- **size_bits** –**[in]** The number of bits required to read.

Returns
- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

esp_err_t **esp_efuse_write_block** (*esp_efuse_block_t* blk, const void *src_key, size_t offset_in_bits, size_t size_bits)

Write key to efuse block starting at the offset and the required size.

Parameters
- **blk** –**[in]** Block number of eFuse.
- **src_key** –**[in]** A pointer to array that contains the key for writing.
- **offset_in_bits** –**[in]** Start bit in block.
- **size_bits** –**[in]** The number of bits required to write.

Returns
- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits

uint32_t **esp_efuse_get_pkg_ver** (void)

Returns chip package from efuse.

Returns chip package

void **esp_efuse_reset**(void)

Reset efuse write registers.

Efuse write registers are written to zero, to negate any changes that have been staged here.

Note: This function is not threadsafe, if calling code updates efuse values from multiple tasks then this is caller's responsibility to serialise.

void **esp_efuse_disable_basic_rom_console**(void)

Disable BASIC ROM Console via efuse.

By default, if booting from flash fails the ESP32 will boot a BASIC console in ROM.

Call this function (from bootloader or app) to permanently disable the console on this chip.

esp_err_t **esp_efuse_disable_rom_download_mode**(void)

Disable ROM Download Mode via eFuse.

Permanently disables the ROM Download Mode feature. Once disabled, if the SoC is booted with strapping pins set for ROM Download Mode then an error is printed instead.

Note: Not all SoCs support this option. An error will be returned if called on an ESP32 with a silicon revision lower than 3, as these revisions do not support this option.

Note: If ROM Download Mode is already disabled, this function does nothing and returns success.

Returns

- ESP_OK If the eFuse was successfully burned, or had already been burned.
- ESP_ERR_NOT_SUPPORTED (ESP32 only) This SoC is not capable of disabling UART download mode
- ESP_ERR_INVALID_STATE (ESP32 only) This eFuse is write protected and cannot be written

esp_err_t **esp_efuse_set_rom_log_scheme**(*esp_efuse_rom_log_scheme_t* log_scheme)

Set boot ROM log scheme via eFuse.

Note: By default, the boot ROM will always print to console. This API can be called to set the log scheme only once per chip, once the value is changed from the default it can't be changed again.

Parameters **log_scheme** – Supported ROM log scheme

Returns

- ESP_OK If the eFuse was successfully burned, or had already been burned.
- ESP_ERR_NOT_SUPPORTED (ESP32 only) This SoC is not capable of setting ROM log scheme
- ESP_ERR_INVALID_STATE This eFuse is write protected or has been burned already

uint32_t **esp_efuse_read_secure_version**(void)

Return secure_version from efuse field.

Returns Secure version from efuse field

bool **esp_efuse_check_secure_version**(uint32_t secure_version)

Check secure_version from app and secure_version and from efuse field.

Parameters **secure_version** – Secure version from app.

Returns

- True: If version of app is equal or more then secure_version from efuse.

esp_err_t **esp_efuse_update_secure_version**(uint32_t secure_version)

Write efuse field by secure_version value.

Update the secure_version value is available if the coding scheme is None. Note: Do not use this function in your applications. This function is called as part of the other API.

Parameters **secure_version** –**[in]** Secure version from app.

Returns

- ESP_OK: Successful.
- ESP_FAIL: secure version of app cannot be set to efuse field.
- ESP_ERR_NOT_SUPPORTED: Anti rollback is not supported with the 3/4 and Repeat coding scheme.

esp_err_t **esp_efuse_batch_write_begin**(void)

Set the batch mode of writing fields.

This mode allows you to write the fields in the batch mode when need to burn several efuses at one time. To enable batch mode call begin() then perform as usually the necessary operations read and write and at the end call commit() to actually burn all written efuses. The batch mode can be used nested. The commit will be done by the last commit() function. The number of begin() functions should be equal to the number of commit() functions.

Note: If batch mode is enabled by the first task, at this time the second task cannot write/read efuses. The second task will wait for the first task to complete the batch operation.

```
// Example of using the batch writing mode.

// set the batch writing mode
esp_efuse_batch_write_begin();

// use any writing functions as usual
esp_efuse_write_field_blob(ESP_EFUSE_...);
esp_efuse_write_field_cnt(ESP_EFUSE_...);
esp_efuse_set_write_protect(EFUSE_BLKx);
esp_efuse_write_reg(EFUSE_BLKx, ...);
esp_efuse_write_block(EFUSE_BLKx, ...);
esp_efuse_write(ESP_EFUSE_1, 3);  // ESP_EFUSE_1 == 1, here we write a new␣
↪value = 3. The changes will be burn by the commit() function.
esp_efuse_read_...(ESP_EFUSE_1);  // this function returns ESP_EFUSE_1 == 1␣
↪because uncommitted changes are not readable, it will be available only␣
↪after commit.
...

// esp_efuse_batch_write APIs can be called recursively.
esp_efuse_batch_write_begin();
esp_efuse_set_write_protect(EFUSE_BLKx);
esp_efuse_batch_write_commit(); // the burn will be skipped here, it will be␣
↪done in the last commit().

...

// Write all of these fields to the efuse registers
esp_efuse_batch_write_commit();
esp_efuse_read_...(ESP_EFUSE_1);  // this function returns ESP_EFUSE_1 == 3.
```

Note: Please note that reading in the batch mode does not show uncommitted changes.

Returns
- ESP_OK: Successful.

esp_err_t **esp_efuse_batch_write_cancel** (void)

Reset the batch mode of writing fields.

It will reset the batch writing mode and any written changes.

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_STATE: Tha batch mode was not set.

esp_err_t **esp_efuse_batch_write_commit** (void)

Writes all prepared data for the batch mode.

Must be called to ensure changes are written to the efuse registers. After this the batch writing mode will be reset.

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_STATE: The deferred writing mode was not set.

bool **esp_efuse_block_is_empty** (*esp_efuse_block_t* block)

Checks that the given block is empty.

Returns
- True: The block is empty.
- False: The block is not empty or was an error.

bool **esp_efuse_get_key_dis_read** (*esp_efuse_block_t* block)

Returns a read protection for the key block.

Parameters **block** – **[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns True: The key block is read protected False: The key block is readable.

esp_err_t **esp_efuse_set_key_dis_read** (*esp_efuse_block_t* block)

Sets a read protection for the key block.

Parameters **block** – **[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

bool **esp_efuse_get_key_dis_write** (*esp_efuse_block_t* block)

Returns a write protection for the key block.

Parameters **block** – **[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns True: The key block is write protected False: The key block is writeable.

esp_err_t **esp_efuse_set_key_dis_write** (*esp_efuse_block_t* block)

Sets a write protection for the key block.

Parameters **block** – **[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

bool **esp_efuse_key_block_unused**(*esp_efuse_block_t* block)

Returns true if the key block is unused, false otherwise.

An unused key block is all zero content, not read or write protected, and has purpose 0 (ESP_EFUSE_KEY_PURPOSE_USER)

Parameters **block** –key block to check.
Returns
- True if key block is unused,
- False if key block is used or the specified block index is not a key block.

bool **esp_efuse_find_purpose**(*esp_efuse_purpose_t* purpose, *esp_efuse_block_t* *block)

Find a key block with the particular purpose set.

Parameters
- **purpose** –**[in]** Purpose to search for.
- **block** –**[out]** Pointer in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX which will be set to the key block if found. Can be NULL, if only need to test the key block exists.

Returns
- True: If found,
- False: If not found (value at block pointer is unchanged).

bool **esp_efuse_get_keypurpose_dis_write**(*esp_efuse_block_t* block)

Returns a write protection of the key purpose field for an efuse key block.

Note: For ESP32: no keypurpose, it returns always True.

Parameters **block** –**[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns True: The key purpose is write protected. False: The key purpose is writeable.

esp_efuse_purpose_t **esp_efuse_get_key_purpose**(*esp_efuse_block_t* block)

Returns the current purpose set for an efuse key block.

Parameters **block** –**[in]** A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
Returns
- Value: If Successful, it returns the value of the purpose related to the given key block.
- ESP_EFUSE_KEY_PURPOSE_MAX: Otherwise.

esp_err_t **esp_efuse_write_key**(*esp_efuse_block_t* block, *esp_efuse_purpose_t* purpose, const void *key, size_t key_size_bytes)

Program a block of key data to an efuse block.

The burn of a key, protection bits, and a purpose happens in batch mode.

Parameters
- **block** –**[in]** Block to read purpose for. Must be in range EFUSE_BLK_KEY0 to EFUSE_BLK_KEY_MAX. Key block must be unused (esp_efuse_key_block_unused).
- **purpose** –**[in]** Purpose to set for this key. Purpose must be already unset.
- **key** –**[in]** Pointer to data to write.
- **key_size_bytes** –**[in]** Bytes length of data to write.

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_INVALID_STATE: Error in efuses state, unused block not found.

- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

esp_err_t **esp_efuse_write_keys** (const *esp_efuse_purpose_t* purposes[], uint8_t keys[][32], unsigned number_of_keys)

Program keys to unused efuse blocks.

The burn of keys, protection bits, and purposes happens in batch mode.

Parameters

- **purposes** –**[in]** Array of purposes (purpose[number_of_keys]).
- **keys** –**[in]** Array of keys (uint8_t keys[number_of_keys][32]). Each key is 32 bytes long.
- **number_of_keys** –**[in]** The number of keys to write (up to 6 keys).

Returns

- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_INVALID_STATE: Error in efuses state, unused block not found.
- ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS: Error not enough unused key blocks available
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

esp_err_t **esp_efuse_check_errors** (void)

Checks eFuse errors in BLOCK0.

It does a BLOCK0 check if eFuse EFUSE_ERR_RST_ENABLE is set. If BLOCK0 has an error, it prints the error and returns ESP_FAIL, which should be treated as esp_restart.

Note: Refers to ESP32-C3 only.

Returns

- ESP_OK: No errors in BLOCK0.
- ESP_FAIL: Error in BLOCK0 requiring reboot.

Structures

struct **esp_efuse_desc_t**

Type definition for an eFuse field.

Public Members

esp_efuse_block_t **efuse_block**

Block of eFuse

uint8_t **bit_start**

Start bit [0..255]

uint16_t **bit_count**

Length of bit field [1..-]

Macros

ESP_ERR_EFUSE

Base error code for efuse api.

ESP_OK_EFUSE_CNT

OK the required number of bits is set.

ESP_ERR_EFUSE_CNT_IS_FULL

Error field is full.

ESP_ERR_EFUSE_REPEATED_PROG

Error repeated programming of programmed bits is strictly forbidden.

ESP_ERR_CODING

Error while a encoding operation.

ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS

Error not enough unused key blocks available

ESP_ERR_DAMAGED_READING

Error. Burn or reset was done during a reading operation leads to damage read data. This error is internal to the efuse component and not returned by any public API.

Enumerations

enum **esp_efuse_rom_log_scheme_t**

Type definition for ROM log scheme.

Values:

enumerator **ESP_EFUSE_ROM_LOG_ALWAYS_ON**

Always enable ROM logging

enumerator **ESP_EFUSE_ROM_LOG_ON_GPIO_LOW**

ROM logging is enabled when specific GPIO level is low during start up

enumerator **ESP_EFUSE_ROM_LOG_ON_GPIO_HIGH**

ROM logging is enabled when specific GPIO level is high during start up

enumerator **ESP_EFUSE_ROM_LOG_ALWAYS_OFF**

Disable ROM logging permanently

2.10.7 Error Codes and Helper Functions

This section lists definitions of common ESP-IDF error codes and several helper functions related to error handling.

For general information about error codes in ESP-IDF, see *Error Handling*.

For the full list of error codes defined in ESP-IDF, see *Error Code Reference*.

API Reference

Header File

- components/esp_common/include/esp_check.h

Macros

ESP_RETURN_ON_ERROR (x, log_tag, format, ...)

Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message and returns. In the future, we want to switch to C++20. We also want to become compatible with clang. Hence, we provide two versions of the following macros. The first one is using the GNU extension ##__VA_ARGS__. The second one is using the C++20 feature **VA_OPT**(,). This allows users to compile their code with standard C++20 enabled instead of the GNU extension. Below C++20, we haven't found any good alternative to using ##__VA_ARGS__. Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message and returns.

ESP_RETURN_ON_ERROR_ISR (x, log_tag, format, ...)

A version of ESP_RETURN_ON_ERROR() macro that can be called from ISR.

ESP_GOTO_ON_ERROR (x, goto_tag, log_tag, format, ...)

Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message, sets the local variable 'ret' to the code, and then exits by jumping to 'goto_tag'.

ESP_GOTO_ON_ERROR_ISR (x, goto_tag, log_tag, format, ...)

A version of ESP_GOTO_ON_ERROR() macro that can be called from ISR.

ESP_RETURN_ON_FALSE (a, err_code, log_tag, format, ...)

Macro which can be used to check the condition. If the condition is not 'true', it prints the message and returns with the supplied 'err_code'.

ESP_RETURN_ON_FALSE_ISR (a, err_code, log_tag, format, ...)

A version of ESP_RETURN_ON_FALSE() macro that can be called from ISR.

ESP_GOTO_ON_FALSE (a, err_code, goto_tag, log_tag, format, ...)

Macro which can be used to check the condition. If the condition is not 'true', it prints the message, sets the local variable 'ret' to the supplied 'err_code', and then exits by jumping to 'goto_tag'.

ESP_GOTO_ON_FALSE_ISR (a, err_code, goto_tag, log_tag, format, ...)

A version of ESP_GOTO_ON_FALSE() macro that can be called from ISR.

Header File

- components/esp_common/include/esp_err.h

Functions

const char ***esp_err_to_name** (*esp_err_t* code)

Returns string for esp_err_t error codes.

This function finds the error code in a pre-generated lookup-table and returns its string representation.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Parameters **code** – esp_err_t error code
Returns string error message

const char ***esp_err_to_name_r** (*esp_err_t* code, char *buf, size_t buflen)

Returns string for esp_err_t and system error codes.

This function finds the error code in a pre-generated lookup-table of esp_err_t errors and returns its string representation. If the error code is not found then it is attempted to be found among system errors.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Parameters

- **code** –esp_err_t error code
- **buf** –**[out]** buffer where the error message should be written
- **buflen** –Size of buffer buf. At most buflen bytes are written into the buf buffer (including the terminating null byte).

Returns buf containing the string error message

Macros

ESP_OK

esp_err_t value indicating success (no error)

ESP_FAIL

Generic esp_err_t code indicating failure

ESP_ERR_NO_MEM

Out of memory

ESP_ERR_INVALID_ARG

Invalid argument

ESP_ERR_INVALID_STATE

Invalid state

ESP_ERR_INVALID_SIZE

Invalid size

ESP_ERR_NOT_FOUND

Requested resource not found

ESP_ERR_NOT_SUPPORTED

Operation or feature not supported

ESP_ERR_TIMEOUT

Operation timed out

ESP_ERR_INVALID_RESPONSE

Received response was invalid

ESP_ERR_INVALID_CRC

CRC or checksum was invalid

ESP_ERR_INVALID_VERSION

Version was invalid

ESP_ERR_INVALID_MAC

MAC address was invalid

ESP_ERR_NOT_FINISHED

There are items remained to retrieve

ESP_ERR_WIFI_BASE

Starting number of WiFi error codes

ESP_ERR_MESH_BASE

Starting number of MESH error codes

ESP_ERR_FLASH_BASE

Starting number of flash error codes

ESP_ERR_HW_CRYPTO_BASE

Starting number of HW cryptography module error codes

ESP_ERR_MEMPROT_BASE

Starting number of Memory Protection API error codes

ESP_ERROR_CHECK(x)

Macro which can be used to check the error code, and terminate the program in case the code is not ESP_OK. Prints the error code, error location, and the failed statement to serial output.

Disabled if assertions are disabled.

ESP_ERROR_CHECK_WITHOUT_ABORT(x)

Macro which can be used to check the error code. Prints the error code, error location, and the failed statement to serial output. In comparison with ESP_ERROR_CHECK(), this prints the same error message but isn't terminating the program.

Type Definitions

typedef int **esp_err_t**

2.10.8 ESP HTTPS OTA

Overview

`esp_https_ota` provides simplified APIs to perform firmware upgrades over HTTPS. It's an abstraction layer over existing OTA APIs.

Application Example

```
esp_err_t do_firmware_upgrade()
{
    esp_http_client_config_t config = {
        .url = CONFIG_FIRMWARE_UPGRADE_URL,
        .cert_pem = (char *)server_cert_pem_start,
    };
    esp_https_ota_config_t ota_config = {
        .http_config = &config,
    };
    esp_err_t ret = esp_https_ota(&ota_config);
    if (ret == ESP_OK) {
```

(continues on next page)

(continued from previous page)

```
        esp_restart();
    } else {
        return ESP_FAIL;
    }
    return ESP_OK;
}
```

Server Verification

Please refer to *ESP-TLS: TLS Server Verification* for more information on server verification. The root certificate (in PEM format) needs to be provided to the `esp_http_client_config_t::cert_pem` member.

Note: The server-endpoint **root** certificate should be used for verification instead of any intermediate ones from the certificate chain. The reason being that the root certificate has the maximum validity and usually remains the same for a long period of time. Users can also use the `ESP x509 Certificate Bundle` feature for verification, which covers most of the trusted root certificates (using the `esp_http_client_config_t::crt_bundle_attach` member).

Partial Image Download over HTTPS

To use partial image download feature, enable `partial_http_download` configuration in `esp_https_ota_config_t`. When this configuration is enabled, firmware image will be downloaded in multiple HTTP requests of specified size. Maximum content length of each request can be specified by setting `max_http_request_size` to required value.

This option is useful while fetching image from a service like AWS S3, where mbedTLS Rx buffer size (*CONFIG_MBEDTLS_SSL_IN_CONTENT_LEN*) can be set to lower value which is not possible without enabling this configuration.

Default value of mbedTLS Rx buffer size is set to 16K. By using partial_http_download with max_http_request_size of 4K, size of mbedTLS Rx buffer can be reduced to 4K. With this configuration, memory saving of around 12K is expected.

Signature Verification

For additional security, signature of OTA firmware images can be verified. For that, refer *Secure OTA Updates Without Secure boot*

Advanced APIs

`esp_https_ota` also provides advanced APIs which can be used if more information and control is needed during the OTA process.

Example that uses advanced ESP_HTTPS_OTA APIs: system/ota/advanced_https_ota.

OTA Upgrades with Pre-Encrypted Firmware

To perform OTA upgrades with Pre-Encrypted Firmware, please enable *CONFIG_ESP_HTTPS_OTA_DECRYPT_CB* in component menuconfig.

Example that performs OTA upgrade with Pre-Encrypted Firmware: system/ota/pre_encrypted_ota.

OTA System Events

ESP HTTPS OTA has various events for which a handler can be triggered by *the Event Loop library* when the particular event occurs. The handler has to be registered using *esp_event_handler_register()*. This helps in event handling for ESP HTTPS OTA. *esp_https_ota_event_t* has all the events which can happen when performing OTA upgrade using ESP HTTPS OTA.

Event Handler Example

```
/* Event handler for catching system events */
static void event_handler(void* arg, esp_event_base_t event_base,
                          int32_t event_id, void* event_data)
{
    if (event_base == ESP_HTTPS_OTA_EVENT) {
        switch (event_id) {
            case ESP_HTTPS_OTA_START:
                ESP_LOGI(TAG, "OTA started");
                break;
            case ESP_HTTPS_OTA_CONNECTED:
                ESP_LOGI(TAG, "Connected to server");
                break;
            case ESP_HTTPS_OTA_GET_IMG_DESC:
                ESP_LOGI(TAG, "Reading Image Description");
                break;
            case ESP_HTTPS_OTA_VERIFY_CHIP_ID:
                ESP_LOGI(TAG, "Verifying chip id of new image: %d", *(esp_
↪chip_id_t *)event_data);
                break;
            case ESP_HTTPS_OTA_DECRYPT_CB:
                ESP_LOGI(TAG, "Callback to decrypt function");
                break;
            case ESP_HTTPS_OTA_WRITE_FLASH:
                ESP_LOGD(TAG, "Writing to flash: %d written", *(int␣
↪*)event_data);
                break;
            case ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION:
                ESP_LOGI(TAG, "Boot partition updated. Next Partition: %d
↪", *(esp_partition_subtype_t *)event_data);
                break;
            case ESP_HTTPS_OTA_FINISH:
                ESP_LOGI(TAG, "OTA finish");
                break;
            case ESP_HTTPS_OTA_ABORT:
                ESP_LOGI(TAG, "OTA abort");
                break;
        }
    }
}
```

Expected data type for different ESP HTTPS OTA events in the system event loop:

- ESP_HTTPS_OTA_START : `NULL`
- ESP_HTTPS_OTA_CONNECTED : `NULL`
- ESP_HTTPS_OTA_GET_IMG_DESC : `NULL`
- ESP_HTTPS_OTA_VERIFY_CHIP_ID : `esp_chip_id_t`
- ESP_HTTPS_OTA_DECRYPT_CB : `NULL`
- ESP_HTTPS_OTA_WRITE_FLASH : `int`
- ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION : `esp_partition_subtype_t`
- ESP_HTTPS_OTA_FINISH : `NULL`
- ESP_HTTPS_OTA_ABORT : `NULL`

API Reference

Header File

- components/esp_https_ota/include/esp_https_ota.h

Functions

esp_err_t **esp_https_ota** (const *esp_https_ota_config_t* *ota_config)

HTTPS OTA Firmware upgrade.

This function allocates HTTPS OTA Firmware upgrade context, establishes HTTPS connection, reads image data from HTTP stream and writes it to OTA partition and finishes HTTPS OTA Firmware upgrade operation. This API supports URL redirection, but if CA cert of URLs differ then it should be appended to `cert_pem` member of `ota_config->http_config`.

Note: This API handles the entire OTA operation, so if this API is being used then no other APIs from `esp_https_ota` component should be called. If more information and control is needed during the HTTPS OTA process, then one can use `esp_https_ota_begin` and subsequent APIs. If this API returns successfully, esp_restart() must be called to boot from the new firmware image.

Parameters **ota_config** –**[in]** pointer to *esp_https_ota_config_t* structure.

Returns

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in esp-idf ' s app_update component.

esp_err_t **esp_https_ota_begin** (const *esp_https_ota_config_t* *ota_config, *esp_https_ota_handle_t* *handle)

Start HTTPS OTA Firmware upgrade.

This function initializes ESP HTTPS OTA context and establishes HTTPS connection. This function must be invoked first. If this function returns successfully, then `esp_https_ota_perform` should be called to continue with the OTA process and there should be a call to `esp_https_ota_finish` on completion of OTA operation or on failure in subsequent operations. This API supports URL redirection, but if CA cert of URLs differ then it should be appended to `cert_pem` member of `http_config`, which is a part of `ota_config`. In case of error, this API explicitly sets `handle` to NULL.

Note: This API is blocking, so setting `is_async` member of `http_config` structure will result in an error.

Parameters

- **ota_config** –**[in]** pointer to *esp_https_ota_config_t* structure
- **handle** –**[out]** pointer to an allocated data of type `esp_https_ota_handle_t` which will be initialised in this function

Returns

- ESP_OK: HTTPS OTA Firmware upgrade context initialised and HTTPS connection established
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument (missing/incorrect config, certificate, etc.)
- For other return codes, refer documentation in app_update component and esp_http_client component in esp-idf.

esp_err_t **esp_https_ota_perform** (*esp_https_ota_handle_t* https_ota_handle)

Read image data from HTTP stream and write it to OTA partition.

This function reads image data from HTTP stream and writes it to OTA partition. This function must be called only if esp_https_ota_begin() returns successfully. This function must be called in a loop since it returns after every HTTP read operation thus giving you the flexibility to stop OTA operation midway.

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure

Returns

- ESP_ERR_HTTPS_OTA_IN_PROGRESS: OTA update is in progress, call this API again to continue.
- ESP_OK: OTA update was successful
- ESP_FAIL: OTA update failed
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_VERSION: Invalid chip revision in image header
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in esp-idf ' s app_update component.

bool **esp_https_ota_is_complete_data_received** (*esp_https_ota_handle_t* https_ota_handle)

Checks if complete data was received or not.

Note: This API can be called just before esp_https_ota_finish() to validate if the complete image was indeed received.

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure

Returns

- false
- true

esp_err_t **esp_https_ota_finish** (*esp_https_ota_handle_t* https_ota_handle)

Clean-up HTTPS OTA Firmware upgrade and close HTTPS connection.

This function closes the HTTP connection and frees the ESP HTTPS OTA context. This function switches the boot partition to the OTA partition containing the new firmware image.

Note: If this API returns successfully, esp_restart() must be called to boot from the new firmware image esp_https_ota_finish should not be called after calling esp_https_ota_abort

---

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure

Returns

- ESP_OK: Clean-up successful
- ESP_ERR_INVALID_STATE
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image

esp_err_t **esp_https_ota_abort** (*esp_https_ota_handle_t* https_ota_handle)

Clean-up HTTPS OTA Firmware upgrade and close HTTPS connection.

This function closes the HTTP connection and frees the ESP HTTPS OTA context.

Note: esp_https_ota_abort should not be called after calling esp_https_ota_finish

---

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure
Returns
- ESP_OK: Clean-up successful
- ESP_ERR_INVALID_STATE: Invalid ESP HTTPS OTA state
- ESP_FAIL: OTA not started
- ESP_ERR_NOT_FOUND: OTA handle not found
- ESP_ERR_INVALID_ARG: Invalid argument

esp_err_t **esp_https_ota_get_img_desc** (*esp_https_ota_handle_t* https_ota_handle, *esp_app_desc_t* *new_app_info)

Reads app description from image header. The app description provides information like the "Firmware version" of the image.

Note: This API can be called only after esp_https_ota_begin() and before esp_https_ota_perform(). Calling this API is not mandatory.

Parameters
- **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure
- **new_app_info** –**[out]** pointer to an allocated *esp_app_desc_t* structure

Returns
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Invalid state to call this API. esp_https_ota_begin() not called yet.
- ESP_FAIL: Failed to read image descriptor
- ESP_OK: Successfully read image descriptor

int **esp_https_ota_get_image_len_read** (*esp_https_ota_handle_t* https_ota_handle)

This function returns OTA image data read so far.

Note: This API should be called only if `esp_https_ota_perform()` has been called atleast once or if `esp_https_ota_get_img_desc` has been called before.

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure
Returns
- -1 On failure
- total bytes read so far

int **esp_https_ota_get_image_size** (*esp_https_ota_handle_t* https_ota_handle)

This function returns OTA image total size.

Note: This API should be called after esp_https_ota_begin() has been already called. This can be used to create some sort of progress indication (in combination with esp_https_ota_get_image_len_read())

Parameters **https_ota_handle** –**[in]** pointer to esp_https_ota_handle_t structure
Returns
- -1 On failure or chunked encoding
- total bytes of image

Structures

struct **esp_https_ota_config_t**

ESP HTTPS OTA configuration.

Public Members

const *esp_http_client_config_t* ***http_config**

ESP HTTP client configuration

http_client_init_cb_t **http_client_init_cb**

Callback after ESP HTTP client is initialised

bool **bulk_flash_erase**

Erase entire flash partition during initialization. By default flash partition is erased during write operation and in chunk of 4K sector size

bool **partial_http_download**

Enable Firmware image to be downloaded over multiple HTTP requests

int **max_http_request_size**

Maximum request size for partial HTTP download

Macros

ESP_ERR_HTTPS_OTA_BASE

ESP_ERR_HTTPS_OTA_IN_PROGRESS

Type Definitions

typedef void ***esp_https_ota_handle_t**

typedef *esp_err_t* (***http_client_init_cb_t**)(*esp_http_client_handle_t*)

Enumerations

enum **esp_https_ota_event_t**

Events generated by OTA process.

Values:

enumerator **ESP_HTTPS_OTA_START**

OTA started

enumerator **ESP_HTTPS_OTA_CONNECTED**

Connected to server

enumerator **ESP_HTTPS_OTA_GET_IMG_DESC**

Read app description from image header

enumerator **ESP_HTTPS_OTA_VERIFY_CHIP_ID**

Verify chip id of new image

enumerator **ESP_HTTPS_OTA_DECRYPT_CB**

Callback to decrypt function

enumerator **ESP_HTTPS_OTA_WRITE_FLASH**

Flash write operation

enumerator **ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION**

Boot partition update after successful ota update

enumerator **ESP_HTTPS_OTA_FINISH**

OTA finished

enumerator **ESP_HTTPS_OTA_ABORT**

OTA aborted

2.10.9 Event Loop Library

Overview

The event loop library allows components to declare events to which other components can register handlers –code which will execute when those events occur. This allows loosely coupled components to attach desired behavior to changes in state of other components without application involvement. For instance, a high level connection handling library may subscribe to events produced by the Wi-Fi subsystem directly and act on those events. This also simplifies event processing by serializing and deferring code execution to another context.

Using `esp_event` APIs

There are two objects of concern for users of this library: events and event loops.

Events are occurrences of note. For example, for Wi-Fi, a successful connection to the access point may be an event. Events are referenced using a two part identifier which are discussed more *here*. Event loops are the vehicle by which events get posted by event sources and handled by event handler functions. These two appear prominently in the event loop library APIs.

Using this library roughly entails the following flow:

1. A user defines a function that should run when an event is posted to a loop. This function is referred to as the event handler. It should have the same signature as *`esp_event_handler_t`*.
2. An event loop is created using *`esp_event_loop_create()`*, which outputs a handle to the loop of type *`esp_event_loop_handle_t`*. Event loops created using this API are referred to as user event loops. There is, however, a special type of event loop called the default event loop which are discussed *here*.
3. Components register event handlers to the loop using *`esp_event_handler_register_with()`*. Handlers can be registered with multiple loops, more on that *here*.
4. Event sources post an event to the loop using *`esp_event_post_to()`*.
5. Components wanting to remove their handlers from being called can do so by unregistering from the loop using *`esp_event_handler_unregister_with()`*.
6. Event loops which are no longer needed can be deleted using *`esp_event_loop_delete()`*.

In code, the flow above may look like as follows:

```
// 1. Define the event handler
void run_on_event(void* handler_arg, esp_event_base_t base, int32_t id, void*␣
↪event_data)
{
    // Event handler logic
```

(continues on next page)

(continued from previous page)

```
}

void app_main()
{
    // 2. A configuration structure of type esp_event_loop_args_t is needed to␣
↪specify the properties of the loop to be
    // created. A handle of type esp_event_loop_handle_t is obtained, which is␣
↪needed by the other APIs to reference the loop
    // to perform their operations on.
    esp_event_loop_args_t loop_args = {
        .queue_size = ...,
        .task_name = ...
        .task_priority = ...,
        .task_stack_size = ...,
        .task_core_id = ...
    };

    esp_event_loop_handle_t loop_handle;

    esp_event_loop_create(&loop_args, &loop_handle);

    // 3. Register event handler defined in (1). MY_EVENT_BASE and MY_EVENT_ID␣
↪specifies a hypothetical
    // event that handler run_on_event should execute on when it gets posted to␣
↪the loop.
    esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_
↪on_event, ...);

    ...

    // 4. Post events to the loop. This queues the event on the event loop. At␣
↪some point in time
    // the event loop executes the event handler registered to the posted event,␣
↪in this case run_on_event.
    // For simplicity sake this example calls esp_event_post_to from app_main, but␣
↪posting can be done from
    // any other tasks (which is the more interesting use case).
    esp_event_post_to(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, ...);

    ...

    // 5. Unregistering an unneeded handler
    esp_event_handler_unregister_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_
↪on_event);

    ...

    // 6. Deleting an unneeded event loop
    esp_event_loop_delete(loop_handle);
}
```

Declaring and defining events

As mentioned previously, events consists of two-part identifiers: the event base and the event ID. The event base identifies an independent group of events; the event ID identifies the event within that group. Think of the event base and event ID as a person's last name and first name, respectively. A last name identifies a family, and the first name identifies a person within that family.

The event loop library provides macros to declare and define the event base easily.

Event base declaration:

```
ESP_EVENT_DECLARE_BASE(EVENT_BASE)
```

Event base definition:

```
ESP_EVENT_DEFINE_BASE(EVENT_BASE)
```

Note: In IDF, the base identifiers for system events are uppercase and are postfixed with `_EVENT`. For example, the base for Wi-Fi events is declared and defined as `WIFI_EVENT`, the ethernet event base `ETHERNET_EVENT`, and so on. The purpose is to have event bases look like constants (although they are global variables considering the defintions of macros `ESP_EVENT_DECLARE_BASE` and `ESP_EVENT_DEFINE_BASE`).

For event ID's, declaring them as enumerations is recommended. Once again, for visibility, these are typically placed in public header files.

Event ID:

```
enum {
    EVENT_ID_1,
    EVENT_ID_2,
    EVENT_ID_3,
    ...
}
```

Default Event Loop

The default event loop is a special type of loop used for system events (Wi-Fi events, for example). The handle for this loop is hidden from the user. The creation, deletion, handler registration/unregistration and posting of events is done through a variant of the APIs for user event loops. The table below enumerates those variants, and the user event loops equivalent.

User Event Loops	Default Event Loops
esp_event_loop_create()	*esp_event_loop_create_default()*
esp_event_loop_delete()	*esp_event_loop_delete_default()*
esp_event_handler_register_with()	*esp_event_handler_register()*
esp_event_handler_unregister_with()	*esp_event_handler_unregister()*
esp_event_post_to()	*esp_event_post()*

If you compare the signatures for both, they are mostly similar except the for the lack of loop handle specification for the default event loop APIs.

Other than the API difference and the special designation to which system events are posted to, there is no difference to how default event loops and user event loops behave. It is even possible for users to post their own events to the default event loop, should the user opt to not create their own loops to save memory.

Notes on Handler Registration

It is possible to register a single handler to multiple events individually, i.e. using multiple calls to *esp_event_handler_register_with()*. For those multiple calls, the specific event base and event ID can be specified with which the handler should execute.

However, in some cases it is desirable for a handler to execute on (1) all events that get posted to a loop or (2) all events of a particular base identifier. This is possible using the special event base identifier `ESP_EVENT_ANY_BASE` and special event ID `ESP_EVENT_ANY_ID`. These special identifiers may be passed as the event base and event ID arguments for *esp_event_handler_register_with()*.

Therefore, the valid arguments to *esp_event_handler_register_with()* are:

1. <event base>, <event ID> - handler executes when the event with base <event base> and event ID <event ID> gets posted to the loop
2. <event base>, ESP_EVENT_ANY_ID - handler executes when any event with base <event base> gets posted to the loop
3. ESP_EVENT_ANY_BASE, ESP_EVENT_ANY_ID - handler executes when any event gets posted to the loop

As an example, suppose the following handler registrations were performed:

```
esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_on_
↪event_1, ...);
esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, ESP_EVENT_ANY_ID, run_
↪on_event_2, ...);
esp_event_handler_register_with(loop_handle, ESP_EVENT_ANY_BASE, ESP_EVENT_ANY_ID,␣
↪run_on_event_3, ...);
```

If the hypothetical event `MY_EVENT_BASE`, `MY_EVENT_ID` is posted, all three handlers `run_on_event_1`, `run_on_event_2`, and `run_on_event_3` would execute.

If the hypothetical event `MY_EVENT_BASE`, `MY_OTHER_EVENT_ID` is posted, only `run_on_event_2` and `run_on_event_3` would execute.

If the hypothetical event `MY_OTHER_EVENT_BASE`, `MY_OTHER_EVENT_ID` is posted, only `run_on_event_3` would execute.

Handler Registration and Handler Dispatch Order The general rule is that for handlers that match a certain posted event during dispatch, those which are registered first also gets executed first. The user can then control which handlers get executed first by registering them before other handlers, provided that all registrations are performed using a single task. If the user plans to take advantage of this behavior, caution must be exercised if there are multiple tasks registering handlers. While the 'first registered, first executed' behavior still holds true, the task which gets executed first will also get their handlers registered first. Handlers registered one after the other by a single task will still be dispatched in the order relative to each other, but if that task gets pre-empted in between registration by another task which also registers handlers; then during dispatch those handlers will also get executed in between.

Event loop profiling

A configuration option *CONFIG_ESP_EVENT_LOOP_PROFILING* can be enabled in order to activate statistics collection for all event loops created. The function `esp_event_dump()` can be used to output the collected statistics to a file stream. More details on the information included in the dump can be found in the `esp_event_dump()` API Reference.

Application Example

Examples on using the `esp_event` library can be found in system/esp_event. The examples cover event declaration, loop creation, handler registration and unregistration and event posting.

Other examples which also adopt esp_event library:

- NMEA Parser , which will decode the statements received from GPS.

API Reference

Header File

- components/esp_event/include/esp_event.h

Functions

esp_err_t **esp_event_loop_create**(const *esp_event_loop_args_t* *event_loop_args, *esp_event_loop_handle_t* *event_loop)

Create a new event loop.

Parameters
- **event_loop_args** –**[in]** configuration structure for the event loop to create
- **event_loop** –**[out]** handle to the created event loop

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: event_loop_args or event_loop was NULL
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- ESP_FAIL: Failed to create task loop
- Others: Fail

esp_err_t **esp_event_loop_delete**(*esp_event_loop_handle_t* event_loop)

Delete an existing event loop.

Parameters **event_loop** –**[in]** event loop to delete, must not be NULL

Returns
- ESP_OK: Success
- Others: Fail

esp_err_t **esp_event_loop_create_default**(void)

Create default event loop.

Returns
- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- ESP_ERR_INVALID_STATE: Default event loop has already been created
- ESP_FAIL: Failed to create task loop
- Others: Fail

esp_err_t **esp_event_loop_delete_default**(void)

Delete the default event loop.

Returns
- ESP_OK: Success
- Others: Fail

esp_err_t **esp_event_loop_run**(*esp_event_loop_handle_t* event_loop, TickType_t ticks_to_run)

Dispatch events posted to an event loop.

This function is used to dispatch events posted to a loop with no dedicated task, i.e. task name was set to NULL in event_loop_args argument during loop creation. This function includes an argument to limit the amount of time it runs, returning control to the caller when that time expires (or some time afterwards). There is no guarantee that a call to this function will exit at exactly the time of expiry. There is also no guarantee that events have been dispatched during the call, as the function might have spent all the allotted time waiting on the event queue. Once an event has been dequeued, however, it is guaranteed to be dispatched. This guarantee contributes to not being able to exit exactly at time of expiry as (1) blocking on internal mutexes is necessary for dispatching the dequeued event, and (2) during dispatch of the dequeued event there is no way to control the time occupied by handler code execution. The guaranteed time of exit is therefore the allotted time + amount of time required to dispatch the last dequeued event.

In cases where waiting on the queue times out, ESP_OK is returned and not ESP_ERR_TIMEOUT, since it is normal behavior.

Note: encountering an unknown event that has been posted to the loop will only generate a warning, not an error.

Parameters

- **event_loop** –**[in]** event loop to dispatch posted events from, must not be NULL
- **ticks_to_run** –**[in]** number of ticks to run the loop

Returns

- ESP_OK: Success
- Others: Fail

esp_err_t **esp_event_handler_register**(esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler, void *event_handler_arg)

Register an event handler to the system event loop (legacy).

This function can be used to register a handler for either: (1) specific events, (2) all events of a certain event base, or (3) all events known by the system event loop.

- specific events: specify exact event_base and event_id
- all events of a certain base: specify exact event_base and use ESP_EVENT_ANY_ID as the event_id
- all events known by the loop: use ESP_EVENT_ANY_BASE for event_base and ESP_EVENT_ANY_ID as the event_id

Registering multiple handlers to events is possible. Registering a single handler to multiple events is also possible. However, registering the same handler to the same event multiple times would cause the previous registrations to be overwritten.

Note: This function is obsolete and will be deprecated soon, please use esp_event_handler_instance_register() instead.

Note: the event loop library does not maintain a copy of event_handler_arg, therefore the user should ensure that event_handler_arg still points to a valid location by the time the handler gets called

Parameters

- **event_base** –**[in]** the base ID of the event to register the handler for
- **event_id** –**[in]** the ID of the event to register the handler for
- **event_handler** –**[in]** the handler function which gets called when the event is dispatched
- **event_handler_arg** –**[in]** data, aside from event data, that is passed to the handler when it is called

Returns

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for the handler
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_handler_register_with**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler, void *event_handler_arg)

Register an event handler to a specific loop (legacy).

This function behaves in the same manner as esp_event_handler_register, except the additional specification of the event loop to register the handler to.

Note: This function is obsolete and will be deprecated soon, please use esp_event_handler_instance_register_with() instead.

Note: the event loop library does not maintain a copy of event_handler_arg, therefore the user should ensure that event_handler_arg still points to a valid location by the time the handler gets called

Parameters

- **event_loop** –**[in]** the event loop to register this handler function to, must not be NULL
- **event_base** –**[in]** the base ID of the event to register the handler for
- **event_id** –**[in]** the ID of the event to register the handler for
- **event_handler** –**[in]** the handler function which gets called when the event is dispatched
- **event_handler_arg** –**[in]** data, aside from event data, that is passed to the handler when it is called

Returns

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for the handler
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_handler_instance_register_with**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler, void *event_handler_arg, *esp_event_handler_instance_t* *instance)

Register an instance of event handler to a specific loop.

This function can be used to register a handler for either: (1) specific events, (2) all events of a certain event base, or (3) all events known by the system event loop.

- specific events: specify exact event_base and event_id
- all events of a certain base: specify exact event_base and use ESP_EVENT_ANY_ID as the event_id
- all events known by the loop: use ESP_EVENT_ANY_BASE for event_base and ESP_EVENT_ANY_ID as the event_id

Besides the error, the function returns an instance object as output parameter to identify each registration. This is necessary to remove (unregister) the registration before the event loop is deleted.

Registering multiple handlers to events, registering a single handler to multiple events as well as registering the same handler to the same event multiple times is possible. Each registration yields a distinct instance object which identifies it over the registration lifetime.

Note: the event loop library does not maintain a copy of event_handler_arg, therefore the user should ensure that event_handler_arg still points to a valid location by the time the handler gets called

Parameters

- **event_loop** –**[in]** the event loop to register this handler function to, must not be NULL
- **event_base** –**[in]** the base ID of the event to register the handler for
- **event_id** –**[in]** the ID of the event to register the handler for
- **event_handler** –**[in]** the handler function which gets called when the event is dispatched
- **event_handler_arg** –**[in]** data, aside from event data, that is passed to the handler when it is called
- **instance** –**[out]** An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same

for all registrations. If no unregistration is needed, but the handler should be deleted when the event loop is deleted, instance can be NULL.

Returns

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for the handler
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID or instance is NULL
- Others: Fail

esp_err_t **esp_event_handler_instance_register**(esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler, void *event_handler_arg, *esp_event_handler_instance_t* *instance)

Register an instance of event handler to the default loop.

This function does the same as esp_event_handler_instance_register_with, except that it registers the handler to the default event loop.

Note: the event loop library does not maintain a copy of event_handler_arg, therefore the user should ensure that event_handler_arg still points to a valid location by the time the handler gets called

---

Parameters

- **event_base** –**[in]** the base ID of the event to register the handler for
- **event_id** –**[in]** the ID of the event to register the handler for
- **event_handler** –**[in]** the handler function which gets called when the event is dispatched
- **event_handler_arg** –**[in]** data, aside from event data, that is passed to the handler when it is called
- **instance** –**[out]** An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same for all registrations. If no unregistration is needed, but the handler should be deleted when the event loop is deleted, instance can be NULL.

Returns

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for the handler
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID or instance is NULL
- Others: Fail

esp_err_t **esp_event_handler_unregister**(esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler)

Unregister a handler with the system event loop (legacy).

Unregisters a handler, so it will no longer be called during dispatch. Handlers can be unregistered for any combination of event_base and event_id which were previously registered. To unregister a handler, the event_base and event_id arguments must match exactly the arguments passed to esp_event_handler_register() when that handler was registered. Passing ESP_EVENT_ANY_BASE and/or ESP_EVENT_ANY_ID will only unregister handlers that were registered with the same wildcard arguments.

Note: This function is obsolete and will be deprecated soon, please use esp_event_handler_instance_unregister() instead.

Note: When using ESP_EVENT_ANY_ID, handlers registered to specific event IDs using the same base will not be unregistered. When using ESP_EVENT_ANY_BASE, events registered to specific bases will also not be unregistered. This avoids accidental unregistration of handlers registered by other users or components.

Parameters

- **event_base** –**[in]** the base of the event with which to unregister the handler
- **event_id** –**[in]** the ID of the event with which to unregister the handler
- **event_handler** –**[in]** the handler to unregister

Returns ESP_OK success

Returns ESP_ERR_INVALID_ARG invalid combination of event base and event ID

Returns others fail

esp_err_t **esp_event_handler_unregister_with**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, *esp_event_handler_t* event_handler)

Unregister a handler from a specific event loop (legacy).

This function behaves in the same manner as esp_event_handler_unregister, except the additional specification of the event loop to unregister the handler with.

Note: This function is obsolete and will be deprecated soon, please use esp_event_handler_instance_unregister_with() instead.

Parameters

- **event_loop** –**[in]** the event loop with which to unregister this handler function, must not be NULL
- **event_base** –**[in]** the base of the event with which to unregister the handler
- **event_id** –**[in]** the ID of the event with which to unregister the handler
- **event_handler** –**[in]** the handler to unregister

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_handler_instance_unregister_with**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, *esp_event_handler_instance_t* instance)

Unregister a handler instance from a specific event loop.

Unregisters a handler instance, so it will no longer be called during dispatch. Handler instances can be unregistered for any combination of event_base and event_id which were previously registered. To unregister a handler instance, the event_base and event_id arguments must match exactly the arguments passed to esp_event_handler_instance_register() when that handler instance was registered. Passing ESP_EVENT_ANY_BASE and/or ESP_EVENT_ANY_ID will only unregister handler instances that were registered with the same wildcard arguments.

Note: When using ESP_EVENT_ANY_ID, handlers registered to specific event IDs using the same base will not be unregistered. When using ESP_EVENT_ANY_BASE, events registered to specific bases will also not be unregistered. This avoids accidental unregistration of handlers registered by other users or components.

Parameters

- **event_loop** –**[in]** the event loop with which to unregister this handler function, must not be NULL
- **event_base** –**[in]** the base of the event with which to unregister the handler
- **event_id** –**[in]** the ID of the event with which to unregister the handler
- **instance** –**[in]** the instance object of the registration to be unregistered

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_handler_instance_unregister**(esp_event_base_t event_base, int32_t event_id, *esp_event_handler_instance_t* instance)

Unregister a handler from the system event loop.

This function does the same as esp_event_handler_instance_unregister_with, except that it unregisters the handler instance from the default event loop.

Parameters

- **event_base** –**[in]** the base of the event with which to unregister the handler
- **event_id** –**[in]** the ID of the event with which to unregister the handler
- **instance** –**[in]** the instance object of the registration to be unregistered

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_post**(esp_event_base_t event_base, int32_t event_id, const void *event_data, size_t event_data_size, TickType_t ticks_to_wait)

Posts an event to the system default event loop. The event loop library keeps a copy of event_data and manages the copy's lifetime automatically (allocation + deletion); this ensures that the data the handler receives is always valid.

Parameters

- **event_base** –**[in]** the event base that identifies the event
- **event_id** –**[in]** the event ID that identifies the event
- **event_data** –**[in]** the data, specific to the event occurrence, that gets passed to the handler
- **event_data_size** –**[in]** the size of the event data
- **ticks_to_wait** –**[in]** number of ticks to block on a full event queue

Returns

- ESP_OK: Success
- ESP_ERR_TIMEOUT: Time to wait for event queue to unblock expired, queue full when posting from ISR
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_post_to**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, const void *event_data, size_t event_data_size, TickType_t ticks_to_wait)

Posts an event to the specified event loop. The event loop library keeps a copy of event_data and manages the copy's lifetime automatically (allocation + deletion); this ensures that the data the handler receives is always valid.

This function behaves in the same manner as esp_event_post_to, except the additional specification of the event loop to post the event to.

Parameters

- **event_loop** –**[in]** the event loop to post to, must not be NULL
- **event_base** –**[in]** the event base that identifies the event
- **event_id** –**[in]** the event ID that identifies the event

- **event_data** –**[in]** the data, specific to the event occurrence, that gets passed to the handler
- **event_data_size** –**[in]** the size of the event data
- **ticks_to_wait** –**[in]** number of ticks to block on a full event queue

Returns

- ESP_OK: Success
- ESP_ERR_TIMEOUT: Time to wait for event queue to unblock expired, queue full when posting from ISR
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

esp_err_t **esp_event_isr_post**(esp_event_base_t event_base, int32_t event_id, const void *event_data, size_t event_data_size, BaseType_t *task_unblocked)

Special variant of esp_event_post for posting events from interrupt handlers.

Note: this function is only available when CONFIG_ESP_EVENT_POST_FROM_ISR is enabled

Note: when this function is called from an interrupt handler placed in IRAM, this function should be placed in IRAM as well by enabling CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Parameters

- **event_base** –**[in]** the event base that identifies the event
- **event_id** –**[in]** the event ID that identifies the event
- **event_data** –**[in]** the data, specific to the event occurrence, that gets passed to the handler
- **event_data_size** –**[in]** the size of the event data; max is 4 bytes
- **task_unblocked** –**[out]** an optional parameter (can be NULL) which indicates that an event task with higher priority than currently running task has been unblocked by the posted event; a context switch should be requested before the interrupt is existed.

Returns

- ESP_OK: Success
- ESP_FAIL: Event queue for the default event loop full
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID, data size of more than 4 bytes
- Others: Fail

esp_err_t **esp_event_isr_post_to**(*esp_event_loop_handle_t* event_loop, esp_event_base_t event_base, int32_t event_id, const void *event_data, size_t event_data_size, BaseType_t *task_unblocked)

Special variant of esp_event_post_to for posting events from interrupt handlers.

Note: this function is only available when CONFIG_ESP_EVENT_POST_FROM_ISR is enabled

Note: when this function is called from an interrupt handler placed in IRAM, this function should be placed in IRAM as well by enabling CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Parameters

- **event_loop** –**[in]** the event loop to post to, must not be NULL
- **event_base** –**[in]** the event base that identifies the event
- **event_id** –**[in]** the event ID that identifies the event
- **event_data** –**[in]** the data, specific to the event occurrence, that gets passed to the handler

- **event_data_size** –**[in]** the size of the event data
- **task_unblocked** –**[out]** an optional parameter (can be NULL) which indicates that an event task with higher priority than currently running task has been unblocked by the posted event; a context switch should be requested before the interrupt is existed.

Returns

- ESP_OK: Success
- ESP_FAIL: Event queue for the loop full
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID, data size of more than 4 bytes
- Others: Fail

esp_err_t **esp_event_dump**(FILE *file)

Dumps statistics of all event loops.

Dumps event loop info in the format:

```
      event loop
          handler
          handler
          ...
      event loop
          handler
          handler
          ...

 where:

  event loop
      format: address,name rx:total_received dr:total_dropped
      where:
          address - memory address of the event loop
          name - name of the event loop, 'none' if no dedicated task
          total_received - number of successfully posted events
          total_dropped - number of events unsuccessfully posted due to queue␣
↪being full

  handler
      format: address ev:base,id inv:total_invoked run:total_runtime
      where:
          address - address of the handler function
          base,id - the event specified by event base and ID this handler␣
↪executes
          total_invoked - number of times this handler has been invoked
          total_runtime - total amount of time used for invoking this handler
```

Note: this function is a noop when CONFIG_ESP_EVENT_LOOP_PROFILING is disabled

Parameters **file** –**[in]** the file stream to output to

Returns

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- Others: Fail

Structures

struct **esp_event_loop_args_t**

Configuration for creating event loops.

Public Members

int32_t **queue_size**

size of the event loop queue

const char ***task_name**

name of the event loop task; if NULL, a dedicated task is not created for event loop

UBaseType_t **task_priority**

priority of the event loop task, ignored if task name is NULL

uint32_t **task_stack_size**

stack size of the event loop task, ignored if task name is NULL

BaseType_t **task_core_id**

core to which the event loop task is pinned to, ignored if task name is NULL

Header File

- components/esp_event/include/esp_event_base.h

Macros

ESP_EVENT_DECLARE_BASE(id)

ESP_EVENT_DEFINE_BASE(id)

ESP_EVENT_ANY_BASE

register handler for any event base

ESP_EVENT_ANY_ID

register handler for any event id

Type Definitions

typedef void ***esp_event_loop_handle_t**

a number that identifies an event with respect to a base

typedef void (***esp_event_handler_t**)(void *event_handler_arg, esp_event_base_t event_base, int32_t event_id, void *event_data)

function called when an event is posted to the queue

typedef void ***esp_event_handler_instance_t**

context identifying an instance of a registered event handler

Related Documents

2.10.10 FreeRTOS (Overview)

Overview

FreeRTOS is an open source real-time operating system kernel that acts as the operating system for ESP-IDF applications and is integrated into ESP-IDF as a component. The FreeRTOS component in ESP-IDF contains ports

of the FreeRTOS kernel for all the CPU architectures used by ESP targets (i.e., Xtensa and RISC-V). Furthermore, ESP-IDF provides different implementations of FreeRTOS in order to support SMP (Symmetric Multiprocessing) on multi-core ESP targets. This document provides an overview of the FreeRTOS component, the FreeRTOS implementations offered by ESP-IDF, and the common aspects across all implementations.

Implementations

The official FreeRTOS (henceforth referred to as Vanilla FreeRTOS) is a single-core RTOS. In order to support the various multi-core ESP targets, ESP-IDF supports different FreeRTOS implementations, namely **ESP-IDF FreeRTOS** and **Amazon SMP FreeRTOS**.

ESP-IDF FreeRTOS ESP-IDF FreeRTOS is a FreeRTOS implementation based on Vanilla FreeRTOS v10.4.3, but contains significant modifications to support SMP. ESP-IDF FreeRTOS only supports two cores at most (i.e., dual core SMP), but is more optimized for this scenario by design. For more details regarding ESP-IDF FreeRTOS and its modifications, please refer to the *FreeRTOS (ESP-IDF)* document.

Note: ESP-IDF FreeRTOS is currently the default FreeRTOS implementation for ESP-IDF.

Amazon SMP FreeRTOS Amazon SMP FreeRTOS is an SMP implementation of FreeRTOS that is officially supported by Amazon. Amazon SMP FreeRTOS is able to support N-cores (i.e., more than two cores). Amazon SMP FreeRTOS can be enabled via the *CONFIG_FREERTOS_SMP* option. For more details regarding Amazon SMP FreeRTOS, please refer to the official Amazon SMP FreeRTOS documentation.

Warning: The Amazon SMP FreeRTOS implementation (and its port in ESP-IDF) are currently in experimental/beta state. Therefore, significant behavioral changes and breaking API changes can occur.

Configuration

Kernel Configuration Vanilla FreeRTOS requires that ports and applications configure the kernel by adding various `#define config...` macros to `FreeRTOSConfig.h`. Vanilla FreeRTOS supports a list of kernel configuration options which allow various kernel behaviors and features to be enabled or disabled.

However, for all FreeRTOS ports in ESP-IDF, the ``FreeRTOSConfig.h`` file is considered private and must not be modified by users. A large number of kernel configuration options in `FreeRTOSConfig.h` are hard coded as they are either required or not supported in ESP-IDF. All kernel configuration options that are configurable by the user will be exposed via menuconfig under `Component Config/FreeRTOS/Kernel`.

For the full list of user configurable kernel options, see *Project Configuration*. The list below highlights some commonly used kernel configuration options:

- *CONFIG_FREERTOS_UNICORE* will run FreeRTOS only on CPU0. Note that this is **not equivalent to running Vanilla FreeRTOS**. Furthermore, this option may affect behavior of components other than freertos. For more details regarding the effects of running FreeRTOS on a single core, refer to *ESP-IDF FreeRTOS Single Core* (if using ESP-IDF FreeRTOS) or the official Amazon SMP FreeRTOS documentation. Alternatively, users can also search for occurrences of `CONFIG_FREERTOS_UNICORE` in the ESP-IDF components.
- *CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY* enables backward compatibility with some FreeRTOS macros/types/functions that were deprecated from v8.0 onwards.

Port Configuration All other FreeRTOS related configuration options that are not part of the kernel configuration are exposed via menuconfig under `Component Config/FreeRTOS/Port`. These options configure aspects such as:

- The FreeRTOS ports themselves (e.g., tick timer selection, ISR stack size)

- Additional features added to the FreeRTOS implementation or ports

Using FreeRTOS

Application Entry Point Unlike Vanilla FreeRTOS, users of FreeRTOS in ESP-IDF **must never call** *vTaskStartScheduler()* and *vTaskEndScheduler()*. Instead, ESP-IDF will start FreeRTOS automatically. Users must define a `void app_main(void)` function which acts as the entry point for user's application and is automatically called on ESP-IDF startup.

- Typically, users would spawn the rest of their application's task from `app_main`.
- The `app_main` function is allowed to return at any point (i.e., before the application terminates).
- The `app_main` function is called from the `main` task.

Background Tasks During startup, ESP-IDF and FreeRTOS will automatically create multiple tasks that run in the background (listed in the the table below).

Table 10: List of Tasks Created During Startup

Task Name	Description	Stack Size	Affinity	Priority
Idle Tasks (`IDLEx`)	An idle task (`IDLEx`) is created for (and pinned to) each CPU core, where `x` is the CPU core's number. The `x` is dropped when single-core configuration is enabled.	*CONFIG_FREERTOS_IDLE_TASK_ST*	CPU`x`	0
FreeRTOS Timer Task (`Tmr Svc`)	FreeRTOS will create the Timer Service/Daemon Task if any FreeRTOS Timer APIs are called by the application.	*CONFIG_FREERTOS_TIMER_TASK_S*	CPU0	*CONFIG_FREERTOS_TIMER_TASK_PRIO*
Main Task (`main`)	Task that simply calls `app_main`. This task will self delete when `app_main` returns	*CONFIG_ESP_MAIN_TASK_STACK_SI*	*CONFIG_ESP_MAIN_TASK_AFFINITY*	1
IPC Tasks (`ipcx`)	When *CONFIG_FREERTOS_UNICORE* is false, an IPC task (`ipcx`) is created for (and pinned to) each CPU. IPC tasks are used to implement the Inter-processor Call (IPC) feature.	*CONFIG_ESP_IPC_TASK_STACK_SIZ*	CPU`x`	24
ESP Timer Task (`esp_timer`)	ESP-IDF will create the ESP Timer Task used to process ESP Timer callbacks.	*CONFIG_ESP_TIMER_TASK_STACK_*	CPU0	22

Note: Note that if an application uses other ESP-IDF features (e.g., WiFi or Bluetooth), those features may create their own background tasks in addition to the tasks listed in the table above.

FreeRTOS Additions

ESP-IDF provides some supplemental features to FreeRTOS such as Ring Buffers, ESP-IDF style Tick and Idle Hooks, and TLSP deletion callbacks. See *FreeRTOS (Supplemental Features)* for more details.

2.10.11 FreeRTOS (ESP-IDF)

Overview

The original FreeRTOS (hereinafter referred to as Vanilla FreeRTOS) is a small and efficient Real Time Operating System supported on many single-core MCUs and SoCs. However, to support numerous dual core ESP targets (such as the ESP32 and ESP32-S3), ESP-IDF provides a dual core SMP (Symmetric Multiprocessing) capable implementation of FreeRTOS, (hereinafter referred to as ESP-IDF FreeRTOS).

ESP-IDF FreeRTOS is based on Vanilla FreeRTOS v10.4.3, but contains significant modifications to both API and kernel behavior in order to support dual core SMP. This document describes the API and behavioral differences between Vanilla FreeRTOS and ESP-IDF FreeRTOS.

Note: This document assumes that the reader has a requisite understanding of Vanilla FreeRTOS (its features, behavior, and API usage). Refer to the Vanilla FreeRTOS documentation for more details.

Note: ESP-IDF FreeRTOS can be built for single core by enabling the *CONFIG_FREERTOS_UNICORE* configuration option. ESP targets that are single core will always have the *CONFIG_FREERTOS_UNICORE* option enabled. However, note that building with *CONFIG_FREERTOS_UNICORE* enabled does not equate to building with Vanilla FreeRTOS (i.e., some of the behavioral and API changes of ESP-IDF will still be present). For more details, see *ESP-IDF FreeRTOS Single Core* for more details.

This document is split into the following parts.

Contents

- *FreeRTOS (ESP-IDF)*
  - *Overview*
  - *Symmetric Multiprocessing*
  - *Tasks*
  - *SMP Scheduler*
  - *Critical Sections*
  - *Misc*
  - *API Reference*

Symmetric Multiprocessing

Basic Concepts SMP (Symmetric Multiprocessing) is a computing architecture where two or more identical CPUs (cores) are connected to a single shared main memory and controlled by a single operating system. In general, an SMP system···

- has multiple cores running independently. Each core has its own register file, interrupts, and interrupt handling.
- presents an identical view of memory to each core. Thus a piece of code that accesses a particular memory address will have the same effect regardless of which core it runs on.

The main advantages of an SMP system compared to single core or Asymmetric Multiprocessing systems are that···

- the presence of multiple CPUs allows for multiple hardware threads, thus increases overall processing throughput.
- having symmetric memory means that threads can switch cores during execution. This in general can lead to better CPU utilization.

Although an SMP system allows threads to switch cores, there are scenarios where a thread must/should only run on a particular core. Therefore, threads in an SMP systems will also have a core affinity that specifies which particular core the thread is allowed to run on.

- A thread that is pinned to a particular core will only be able to run on that core

- A thread that is unpinned will be allowed to switch between cores during execution instead of being pinned to a particular core.

SMP on an ESP Target ESP targets (such as the ESP32, ESP32-S3) are dual core SMP SoCs. These targets have the following hardware features that make them SMP capable:

- Two identical cores known as CPU0 (i.e., Protocol CPU or PRO_CPU) and CPU1 (i.e., Application CPU or APP_CPU). This means that the execution of a piece of code is identical regardless of which core it runs on.
- Symmetric memory (with some small exceptions).
  - If multiple cores access the same memory address, their access will be serialized at the memory bus level.
  - True atomic access to the same memory address is achieved via an atomic compare-and-swap instruction provided by the ISA.
- Cross-core interrupts that allow one CPU to trigger and interrupt on another CPU. This allows cores to signal each other.

Note: The "PRO_CPU" and "APP_CPU" aliases for CPU0 and CPU1 exist in ESP-IDF as they reflect how typical IDF applications will utilize the two CPUs. Typically, the tasks responsible for handling wireless networking (e.g., WiFi or Bluetooth) will be pinned to CPU0 (thus the name PRO_CPU), whereas the tasks handling the remainder of the application will be pinned to CPU1 (thus the name APP_CPU).

Tasks

Creation Vanilla FreeRTOS provides the following functions to create a task:

- *`xTaskCreate()`* creates a task. The task's memory is dynamically allocated
- *`xTaskCreateStatic()`* creates a task. The task's memory is statically allocated (i.e., provided by the user)

However, in an SMP system, tasks need to be assigned a particular affinity. Therefore, ESP-IDF provides a `PinnedToCore` version of Vanilla FreeRTOS's task creation functions:

- *`xTaskCreatePinnedToCore()`* creates a task with a particular core affinity. The task's memory is dynamically allocated.
- *`xTaskCreateStaticPinnedToCore()`* creates a task with a particular core affinity. The task's memory is statically allocated (i.e., provided by the user)

The `PinnedToCore` versions of the task creation functions API differ from their vanilla counter parts by having an extra `xCoreID` parameter that is used to specify the created task's core affinity. The valid values for core affinity are:

- `0` which pins the created task to CPU0
- `1` which pins the created task to CPU1
- `tskNO_AFFINITY` which allows the task to be run on both CPUs

Note that ESP-IDF FreeRTOS still supports the vanilla versions of the task creation functions. However, they have been modified to simply call their `PinnedToCore` counterparts with `tskNO_AFFINITY`.

Note: ESP-IDF FreeRTOS also changes the units of `ulStackDepth` in the task creation functions. Task stack sizes in Vanilla FreeRTOS are specified in number of words, whereas in ESP-IDF FreeRTOS, the task stack sizes are specified in bytes.

Execution The anatomy of a task in ESP-IDF FreeRTOS is the same as Vanilla FreeRTOS. More specifically, ESP-IDF FreeRTOS tasks:

- Can only be in one of following states: Running, Ready, Blocked, or Suspended.
- Task functions are typically implemented as an infinite loop
- Task functions should never return

Deletion Task deletion in Vanilla FreeRTOS is called via *vTaskDelete()*. The function allows deletion of another task or the currently running task (if the provided task handle is NULL). The actual freeing of the task's memory is sometimes delegated to the idle task (if the task being deleted is the currently running task).

ESP-IDF FreeRTOS provides the same *vTaskDelete()* function. However, due to the dual core nature, there are some behavioral differences when calling *vTaskDelete()* in ESP-IDF FreeRTOS:

- When deleting a task that is pinned to the other core, that task's memory is always freed by the idle task of the other core (due to the need to clear FPU registers).
- When deleting a task that is currently running on the other core, a yield is triggered on the other core and the task's memory is freed by one of the idle tasks (depending on the task's core affinity)
- A deleted task's memory is freed immediately if…
  - The tasks is currently running on this core and is also pinned to this core
  - The task is not currently running and is not pinned to any core

Users should avoid calling *vTaskDelete()* on a task that is currently running on the other core. This is due to the fact that it is difficult to know what the task currently running on the other core is executing, thus can lead to unpredictable behavior such as…

- Deleting a task that is holding a mutex
- Deleting a task that has yet to free memory it previously allocated

Where possible, users should design their application such that *vTaskDelete()* is only ever called on tasks in a known state. For example:

- Tasks self deleting (via `vTaskDelete(NULL)`) when their execution is complete and have also cleaned up all resources used within the task.
- Tasks placing themselves in the suspend state (via *vTaskSuspend()*) before being deleted by another task.

SMP Scheduler

The Vanilla FreeRTOS scheduler is best described as a **Fixed Priority Preemptive scheduler with Time Slicing** meaning that:

- Each tasks is given a constant priority upon creation. The scheduler executes highest priority ready state task
- The scheduler can switch execution to another task without the cooperation of the currently running task
- The scheduler will periodically switch execution between ready state tasks of the same priority (in a round robin fashion). Time slicing is governed by a tick interrupt.

The ESP-IDF FreeRTOS scheduler supports the same scheduling features (i.e., Fixed Priority, Preemption, and Time Slicing) albeit with some small behavioral differences.

Fixed Priority In Vanilla FreeRTOS, when scheduler selects a new task to run, it will always select the current highest priority ready state task. In ESP-IDF FreeRTOS, each core will independently schedule tasks to run. When a particular core selects a task, the core will select the highest priority ready state task that can be run by the core. A task can be run by the core if:

- The task has a compatible affinity (i.e., is either pinned to that core or is unpinned)
- The task is not currently being run by another core

However, users should not assume that the two highest priority ready state tasks are always run by the scheduler as a task's core affinity must also be accounted for. For example, given the following tasks:

- Task A of priority 10 pinned to CPU0
- Task B of priority 9 pinned to CPU0
- Task C of priority 8 pinned to CPU1

The resulting schedule will have Task A running on CPU0 and Task C running on CPU1. Task B is not run even though it is the second highest priority task.

Preemption In Vanilla FreeRTOS, the scheduler can preempt the currently running task if a higher priority task becomes ready to execute. Likewise in ESP-IDF FreeRTOS, each core can be individually preempted by the scheduler if the scheduler determines that a higher priority task can run on that core.

However, there are some instances where a higher priority task that becomes ready can be run on multiple cores. In this case, the scheduler will only preempt one core. The scheduler always gives preference to the current core when multiple cores can be preempted. In other words, if the higher priority ready task is unpinned and has a higher priority than the current priority of both cores, the scheduler will always choose to preempt the current core. For example, given the following tasks:

- Task A of priority 8 currently running on CPU0
- Task B of priority 9 currently running on CPU1
- Task C of priority 10 that is unpinned and was unblocked by Task B

The resulting schedule will have Task A running on CPU0 and Task C preempting Task B given that the scheduler always gives preference to the current core.

Time Slicing The Vanilla FreeRTOS scheduler implements time slicing meaning that if current highest ready priority contains multiple ready tasks, the scheduler will switch between those tasks periodically in a round robin fashion.

However, in ESP-IDF FreeRTOS, it is not possible to implement perfect Round Robin time slicing due to the fact that a particular task may not be able to run on a particular core due to the following reasons:

- The task is pinned to the another core.
- For unpinned tasks, the task is already being run by another core.

Therefore, when a core searches the ready state task list for a task to run, the core may need to skip over a few tasks in the same priority list or drop to a lower priority in order to find a ready state task that the core can run.

The ESP-IDF FreeRTOS scheduler implements a Best Effort Round Robin time slicing for ready state tasks of the same priority by ensuring that tasks that have been selected to run will be placed at the back of the list, thus giving unselected tasks a higher priority on the next scheduling iteration (i.e., the next tick interrupt or yield)

The following example demonstrates the Best Effort Round Robin time slicing in action. Assume that:

- There are four ready state tasks of the same priority `AX, B0, C1, D1` where: - The priority is the current highest priority with ready state tasks - The first character represents the task's names (i.e., `A, B, C, D`) - And the second character represents the tasks core pinning (and `X` means unpinned)
- The task list is always searched from the head

```
--------------------------------------------------------------------------------

1. Starting state. None of the ready state tasks have been selected to run

Head [ AX , B0 , C1 , D0 ] Tail

--------------------------------------------------------------------------------

2. Core 0 has tick interrupt and searches for a task to run.
  Task A is selected and is moved to the back of the list

Core0--|
Head [ AX , B0 , C1 , D0 ] Tail

                        0
Head [ B0 , C1 , D0 , AX ] Tail

--------------------------------------------------------------------------------

3. Core 1 has a tick interrupt and searches for a task to run.
  Task B cannot be run due to incompatible affinity, so core 1 skips to Task C.
  Task C is selected and is moved to the back of the list
```

(continues on next page)

(continued from previous page)

```
Core1-------|          0
Head [ B0 , C1 , D0 , AX ] Tail

                   0    1
Head [ B0 , D0 , AX , C1 ] Tail

-------------------------------------------------------------------------------

4. Core 0 has another tick interrupt and searches for a task to run.
  Task B is selected and moved to the back of the list


Core0--|               1
Head [ B0 , D0 , AX , C1 ] Tail

                   1    0
Head [ D0 , AX , C1 , B0 ] Tail

-------------------------------------------------------------------------------

5. Core 1 has another tick and searches for a task to run.
  Task D cannot be run due to incompatible affinity, so core 1 skips to Task A
  Task A is selected and moved to the back of the list

Core1-------|          0
Head [ D0 , AX , C1 , B0 ] Tail

                   0    1
Head [ D0 , C1 , B0 , AX ] Tail
```

The implications to users regarding the Best Effort Round Robin time slicing:

- Users cannot expect multiple ready state tasks of the same priority to run sequentially (as is the case in Vanilla FreeRTOS). As demonstrated in the example above, a core may need to skip over tasks.
- However, given enough ticks, a task will eventually be given some processing time.
- If a core cannot find a task runnable task at the highest ready state priority, it will drop to a lower priority to search for tasks.
- To achieve ideal round robin time slicing, users should ensure that all tasks of a particular priority are pinned to the same core.

Tick Interrupts Vanilla FreeRTOS requires that a periodic tick interrupt occurs. The tick interrupt is responsible for:

- Incrementing the scheduler' s tick count
- Unblocking any blocked tasks that have timed out
- Checking if time slicing is required (i.e., triggering a context switch)
- Executing the application tick hook

In ESP-IDF FreeRTOS, each core will receive a periodic interrupt and independently run the tick interrupt. The tick interrupts on each core are of the same period but can be out of phase. However, the tick responsibilities listed above are not run by all cores:

- CPU0 will execute all of the tick interrupt responsibilities listed above
- CPU1 will only check for time slicing and execute the application tick hook

---

Note: CPU0 is solely responsible for keeping time in ESP-IDF FreeRTOS. Therefore anything that prevents CPU0 from incrementing the tick count (such as suspending the scheduler on CPU0) will cause the entire schedulers time keeping to lag behind.

Idle Tasks Vanilla FreeRTOS will implicitly create an idle task of priority 0 when the scheduler is started. The idle task runs when no other task is ready to run, and it has the following responsibilities:

- Freeing the memory of deleted tasks
- Executing the application idle hook

In ESP-IDF FreeRTOS, a separate pinned idle task is created for each core. The idle tasks on each core have the same responsibilities as their vanilla counterparts.

Scheduler Suspension Vanilla FreeRTOS allows the scheduler to be suspended/resumed by calling `vTaskSuspendAll()` and `xTaskResumeAll()` respectively. While the scheduler is suspended:

- Task switching is disabled but interrupts are left enabled.
- Calling any blocking/yielding function is forbidden, and time slicing is disabled.
- The tick count is frozen (but the tick interrupt will still occur to execute the application tick hook)

On scheduler resumption, `xTaskResumeAll()` will catch up all of the lost ticks and unblock any timed out tasks.

In ESP-IDF FreeRTOS, suspending the scheduler across multiple cores is not possible. Therefore when `vTaskSuspendAll()` is called on a particular core (e.g., core A):

- Task switching is disabled only on core A but interrupts for core A are left enabled
- Calling any blocking/yielding function on core A is forbidden. Time slicing is disabled on core A.
- If an interrupt on core A unblocks any tasks, those tasks will go into core A's own pending ready task list
- If core A is CPU0, the tick count is frozen and a pended tick count is incremented instead. However, the tick interrupt will still occur in order to execute the application tick hook.

When `xTaskResumeAll()` is called on a particular core (e.g., core A):

- Any tasks added to core A's pending ready task list will be resumed
- If core A is CPU0, the pended tick count is unwound to catch up the lost ticks.

Warning: Given that scheduler suspension on ESP-IDF FreeRTOS will only suspend scheduling on a particular core, scheduler suspension is **NOT** a valid method ensuring mutual exclusion between tasks when accessing shared data. Users should use proper locking primitives such as mutexes or spinlocks if they require mutual exclusion.

Disabling Interrupts Vanilla FreeRTOS allows interrupts to be disabled and enabled by calling `taskDISABLE_INTERRUPTS` and `taskENABLE_INTERRUPTS` respectively.

ESP-IDF FreeRTOS provides the same API, however interrupts will only disabled or enabled on the current core.

Warning: Disabling interrupts is a valid method of achieve mutual exclusion in Vanilla FreeRTOS (and single core systems in general). However, in an SMP system, disabling interrupts is **NOT** a valid method ensuring mutual exclusion. Refer to Critical Sections for more details.

Critical Sections

API Changes Vanilla FreeRTOS implements critical sections by disabling interrupts, This prevents preemptive context switches and the servicing of ISRs during a critical section. Thus a task/ISR that enters a critical section is guaranteed to be the sole entity to access a shared resource. Critical sections in Vanilla FreeRTOS have the following API:

- `taskENTER_CRITICAL()` enters a critical section by disabling interrupts
- `taskEXIT_CRITICAL()` exits a critical section by reenabling interrupts
- `taskENTER_CRITICAL_FROM_ISR()` enters a critical section from an ISR by disabling interrupt nesting
- `taskEXIT_CRITICAL_FROM_ISR()` exits a critical section from an ISR by reenabling interrupt nesting

However, in an SMP system, merely disabling interrupts does not constitute a critical section as the presence of other cores means that a shared resource can still be concurrently accessed. Therefore, critical sections in ESP-IDF FreeRTOS are implemented using spinlocks. To accommodate the spinlocks, the ESP-IDF FreeRTOS critical section APIs contain an additional spinlock parameter as shown below:

- Spinlocks are of `portMUX_TYPE` (**not to be confused to FreeRTOS mutexes**)
- `taskENTER_CRITICAL(&mux)` enters a critical from a task context
- `taskEXIT_CRITICAL(&mux)` exits a critical section from a task context
- `taskENTER_CRITICAL_ISR(&mux)` enters a critical section from an interrupt context
- `taskEXIT_CRITICAL_ISR(&mux)` exits a critical section from an interrupt context

Note: The critical section API can be called recursively (i.e., nested critical sections). Entering a critical section multiple times recursively is valid so long as the critical section is exited the same number of times it was entered. However, given that critical sections can target different spinlocks, users should take care to avoid dead locking when entering critical sections recursively.

Implementation In ESP-IDF FreeRTOS, the process of a particular core entering and exiting a critical section is as follows:

- For `taskENTER_CRITICAL(&mux)` (or `taskENTER_CRITICAL_ISR(&mux)`)
  1. The core disables its interrupts (or interrupt nesting) up to `configMAX_SYSCALL_INTERRUPT_PRIORITY`
  2. The core then spins on the spinlock using an atomic compare-and-set instruction until it acquires the lock. A lock is acquired when the core is able to set the lock's owner value to the core's ID.
  3. Once the spinlock is acquired, the function returns. The remainder of the critical section runs with interrupts (or interrupt nesting) disabled.
- For `taskEXIT_CRITICAL(&mux)` (or `taskEXIT_CRITICAL_ISR(&mux)`)
  1. The core releases the spinlock by clearing the spinlock's owner value
  2. The core re-enables interrupts (or interrupt nesting)

Restrictions and Considerations Given that interrupts (or interrupt nesting) are disabled during a critical section, there are multiple restrictions regarding what can be done within a critical sections. During a critical section, users should keep the following restrictions and considerations in mind:

- Critical sections should be as kept as short as possible
  - The longer the critical section lasts, the longer a pending interrupt can be delayed.
  - A typical critical section should only access a few data structures and/or hardware registers
  - If possible, defer as much processing and/or event handling to the outside of critical sections.
- FreeRTOS API should not be called from within a critical section
- Users should never call any blocking or yielding functions within a critical section

Misc

Floating Point Usage Usually, when a context switch occurs:

- the current state of a CPU's registers are saved to the stack of task being switch out
- the previously saved state of the CPU's registers are loaded from the stack of the task being switched in

However, ESP-IDF FreeRTOS implements Lazy Context Switching for the FPU (Floating Point Unit) registers of a CPU. In other words, when a context switch occurs on a particular core (e.g., CPU0), the state of the core's FPU registers are not immediately saved to the stack of the task getting switched out (e.g., Task A). The FPU's registers are left untouched until:

- A different task (e.g., Task B) runs on the same core and uses the FPU. This will trigger an exception that will save the FPU registers to Task A's stack.
- Task A get's scheduled to the same core and continues execution. Saving and restoring the FPU's registers is not necessary in this case.

However, given that tasks can be unpinned thus can be scheduled on different cores (e.g., Task A switches to CPU1), it is unfeasible to copy and restore the FPU's registers across cores. Therefore, when a task utilizes the FPU (by using a `float` type in its call flow), ESP-IDF FreeRTOS will automatically pin the task to the current core it is running on. This ensures that all tasks that uses the FPU are always pinned to a particular core.

Furthermore, ESP-IDF FreeRTOS by default does not support the usage of the FPU within an interrupt context given that the FPU's register state is tied to a particular task.

Note: ESP targets that contain an FPU do not support hardware acceleration for double precision floating point arithmetic (`double`). Instead `double` is implemented via software hence the behavioral restrictions regarding the `float` type do not apply to `double`. Note that due to the lack of hardware acceleration, `double` operations may consume significantly more CPU time in comparison to `float`.

ESP-IDF FreeRTOS Single Core Although ESP-IDF FreeRTOS is an SMP scheduler, some ESP targets are single core (such as the ESP32-S2 and ESP32-C3). When building ESP-IDF applications for these targets, ESP-IDF FreeRTOS is still used but the number of cores will be set to *1* (i.e., the *CONFIG_FREERTOS_UNICORE* will always be enabled for single core targets).

For multicore targets (such as the ESP32 and ESP32-S3), *CONFIG_FREERTOS_UNICORE* can also be set. This will result in ESP-IDF FreeRTOS only running on CPU0, and all other cores will be inactive.

Note: Users should bear in mind that enabling *CONFIG_FREERTOS_UNICORE* **is NOT equivalent to running Vanilla FreeRTOS**. The additional API of ESP-IDF FreeRTOS can still be called, and the behavior changes of ESP-IDF FreeRTOS will incur a small amount of overhead even when compiled for only a single core.

API Reference

This section contains documentation of FreeRTOS types, functions, and macros. It is automatically generated from FreeRTOS header files.

Task API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/task.h

Functions

BaseType_t **xTaskCreatePinnedToCore**(TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, *TaskHandle_t* *const pvCreatedTask, const BaseType_t xCoreID)

Create a new task with a specified affinity.

This function is similar to xTaskCreate, but allows setting task affinity in SMP system.

Parameters

- **pvTaskCode** –Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop), or should be terminated using vTaskDelete function.
- **pcName** –A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.
- **usStackDepth** –The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- **pvParameters** –Pointer that will be used as the parameter for the task being created.

- **uxPriority** –The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
- **pvCreatedTask** –Used to pass back a handle by which the created task can be referenced.
- **xCoreID** –If the value is tskNO_AFFINITY, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Values 0 or 1 indicate the index number of the CPU which the task should be pinned to. Specifying values larger than (portNUM_PROCESSORS - 1) will cause the function to fail.

Returns pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

static inline BaseType_t **xTaskCreate**(TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, *TaskHandle_t* *const pxCreatedTask)

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreate() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see https://www.FreeRTOS.org/a00111.html). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

See xTaskCreateStatic() for a version that does not use any dynamic memory allocation.

xTaskCreate() can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreateRestricted().

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
  for( ;; )
  {
      // Task code goes here.
  }
}

// Function that creates a task.
void vOtherFunction( void )
{
static uint8_t ucParameterToPass;
TaskHandle_t xHandle = NULL;

  // Create the task, storing the handle.  Note that the passed parameter␣
↪ucParameterToPass
  // must exist for the lifetime of the task, so in this case is declared␣
↪static.  If it was just an
  // an automatic stack variable it might no longer exist, or at least have␣
↪been corrupted, by the time
  // the new task attempts to access it.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_
↪PRIORITY, &xHandle );
  configASSERT( xHandle );

  // Use the handle to delete the task.
  if( xHandle != NULL )
```

(continues on next page)

(continued from previous page)

```
  {
     vTaskDelete( xHandle );
  }
}
```

Note: If program uses thread local variables (ones specified with "__thread" keyword) then storage for them will be allocated on the task's stack.

Parameters

- **pvTaskCode** –Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop), or should be terminated using vTaskDelete function.
- **pcName** –A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.
- **usStackDepth** –The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- **pvParameters** –Pointer that will be used as the parameter for the task being created.
- **uxPriority** –The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
- **pxCreatedTask** –Used to pass back a handle by which the created task can be referenced.

Returns pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

TaskHandle_t **xTaskCreateStaticPinnedToCore**(TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t ulStackDepth, void *const pvParameters, UBaseType_t uxPriority, StackType_t *const pxStackBuffer, StaticTask_t *const pxTaskBuffer, const BaseType_t xCoreID)

Create a new task with a specified affinity.

This function is similar to xTaskCreateStatic, but allows specifying task affinity in an SMP system.

Parameters

- **pvTaskCode** –Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop), or should be terminated using vTaskDelete function.
- **pcName** –A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
- **ulStackDepth** –The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- **pvParameters** –Pointer that will be used as the parameter for the task being created.
- **uxPriority** –The priority at which the task will run.
- **pxStackBuffer** –Must point to a StackType_t array that has at least ulStackDepth indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
- **pxTaskBuffer** –Must point to a variable of type StaticTask_t, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.
- **xCoreID** –If the value is tskNO_AFFINITY, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Values 0 or 1 indicate the index number of the CPU which the task should be pinned to. Specifying values larger than (portNUM_PROCESSORS - 1) will cause the function to fail.

Returns If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and pdPASS is returned. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not

be created and errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned.

static inline *TaskHandle_t* **xTaskCreateStatic**(TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t ulStackDepth, void *const pvParameters, UBaseType_t uxPriority, StackType_t *const puxStackBuffer, StaticTask_t *const pxTaskBuffer)

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreate() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see http://www.freertos.org/a00111.html). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

Example usage:

```
 // Dimensions the buffer that the task being created will use as its stack.
 // NOTE:  This is the number of bytes the stack will hold, not the number of
 // words as found in vanilla FreeRTOS.
#define STACK_SIZE 200

 // Structure that will hold the TCB of the task being created.
 StaticTask_t xTaskBuffer;

 // Buffer that the task being created will use as its stack.  Note this is
 // an array of StackType_t variables.  The size of StackType_t is dependent on
 // the RTOS port.
 StackType_t xStack[ STACK_SIZE ];

 // Function that implements the task being created.
 void vTaskCode( void * pvParameters )
 {
     // The parameter value is expected to be 1 as 1 is passed in the
     // pvParameters value in the call to xTaskCreateStatic().
     configASSERT( ( uint32_t ) pvParameters == 1UL );

     for( ;; )
     {
         // Task code goes here.
     }
 }

 // Function that creates a task.
 void vOtherFunction( void )
 {
     TaskHandle_t xHandle = NULL;

     // Create the task without using any dynamic memory allocation.
     xHandle = xTaskCreateStatic(
                   vTaskCode,       // Function that implements the task.
                   "NAME",          // Text name for the task.
                   STACK_SIZE,      // Stack size in bytes, not words.
                   ( void * ) 1,    // Parameter passed into the task.
                   tskIDLE_PRIORITY,// Priority at which the task is created.
                   xStack,          // Array to use as the task's stack.
                   &xTaskBuffer );  // Variable to hold the task's data␣
↪structure.

     // puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
     // been created, and xHandle will be the task's handle.  Use the handle
```

(continues on next page)

(continued from previous page)

```
    // to suspend the task.
    vTaskSuspend( xHandle );
}
```

Note: If program uses thread local variables (ones specified with "__thread" keyword) then storage for them will be allocated on the task's stack.

Parameters

- **pvTaskCode** –Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop), or should be terminated using vTaskDelete function.
- **pcName** –A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
- **ulStackDepth** –The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- **pvParameters** –Pointer that will be used as the parameter for the task being created.
- **uxPriority** –The priority at which the task will run.
- **puxStackBuffer** –Must point to a StackType_t array that has at least ulStackDepth indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
- **pxTaskBuffer** –Must point to a variable of type StaticTask_t, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.

Returns If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and pdPASS is returned. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not be created and errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned.

BaseType_t **xTaskCreateRestricted**(const TaskParameters_t *const pxTaskDefinition, *TaskHandle_t* *pxCreatedTask)

Only available when configSUPPORT_DYNAMIC_ALLOCATION is set to 1.

xTaskCreateRestricted() should only be used in systems that include an MPU implementation.

Create a new task and add it to the list of tasks that are ready to run. The function parameters define the memory regions and associated access permissions allocated to the task.

See xTaskCreateRestrictedStatic() for a version that does not use any dynamic memory allocation.

return pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Example usage:

```
// Create an TaskParameters_t structure that defines the task to be created.
static const TaskParameters_t xCheckTaskParameters =
{
 vATask,      // pvTaskCode - the function that implements the task.
 "ATask",     // pcName - just a text name for the task to assist debugging.
 100,         // usStackDepth - the stack size DEFINED IN WORDS.
 NULL,        // pvParameters - passed into the task function as the function␣
↪parameters.
 ( 1UL | portPRIVILEGE_BIT ),// uxPriority - task priority, set the␣
↪portPRIVILEGE_BIT if the task should run in a privileged state.
 cStackBuffer,// puxStackBuffer - the buffer to be used as the task stack.

 // xRegions - Allocate up to three separate memory regions for access by
```

(continues on next page)

(continued from previous page)

```
 // the task, with appropriate access permissions.  Different processors have
 // different memory alignment requirements - refer to the FreeRTOS documentation
 // for full information.
 {
     // Base address                 Length  Parameters
     { cReadWriteArray,              32,     portMPU_REGION_READ_WRITE },
     { cReadOnlyArray,               32,     portMPU_REGION_READ_ONLY },
     { cPrivilegedOnlyAccessArray,   128,    portMPU_REGION_PRIVILEGED_READ_WRITE }
 }
};

int main( void )
{
TaskHandle_t xHandle;

 // Create a task from the const structure defined above.  The task handle
 // is requested (the second parameter is not NULL) but in this case just for
 // demonstration purposes as its not actually used.
 xTaskCreateRestricted( &xRegTest1Parameters, &xHandle );

 // Start the scheduler.
 vTaskStartScheduler();

 // Will only get here if there was insufficient memory to create the idle
 // and/or timer task.
 for( ;; );
}
```

Parameters

- **pxTaskDefinition** –Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions.
- **pxCreatedTask** –Used to pass back a handle by which the created task can be referenced.

void **vTaskAllocateMPURegions**(*TaskHandle_t* xTask, const MemoryRegion_t *const pxRegions)

Only available when configSUPPORT_STATIC_ALLOCATION is set to 1.

xTaskCreateRestrictedStatic() should only be used in systems that include an MPU implementation.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreateRestricted() then the stack is provided by the application writer, and the memory used to hold the task's data structure is automatically dynamically allocated inside the xTaskCreateRestricted() function. If a task is created using xTaskCreateRestrictedStatic() then the application writer must provide the memory used to hold the task's data structures too. xTaskCreateRestrictedStatic() therefore allows a memory protected task to be created without using any dynamic memory allocation.

return pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Example usage:

```
// Create an TaskParameters_t structure that defines the task to be created.
// The StaticTask_t variable is only included in the structure when
// configSUPPORT_STATIC_ALLOCATION is set to 1.  The PRIVILEGED_DATA macro can
// be used to force the variable into the RTOS kernel's privileged data area.
```

(continues on next page)

(continued from previous page)

```
static PRIVILEGED_DATA StaticTask_t xTaskBuffer;
static const TaskParameters_t xCheckTaskParameters =
{
 vATask,      // pvTaskCode - the function that implements the task.
 "ATask",     // pcName - just a text name for the task to assist debugging.
  100,        // usStackDepth - the stack size DEFINED IN BYTES.
 NULL,        // pvParameters - passed into the task function as the function␣
↪parameters.
 ( 1UL | portPRIVILEGE_BIT ),// uxPriority - task priority, set the␣
↪portPRIVILEGE_BIT if the task should run in a privileged state.
 cStackBuffer,// puxStackBuffer - the buffer to be used as the task stack.

 // xRegions - Allocate up to three separate memory regions for access by
 // the task, with appropriate access permissions.  Different processors have
 // different memory alignment requirements - refer to the FreeRTOS␣
↪documentation
 // for full information.
 {
     // Base address                 Length  Parameters
     { cReadWriteArray,              32,     portMPU_REGION_READ_WRITE },
     { cReadOnlyArray,               32,     portMPU_REGION_READ_ONLY },
     { cPrivilegedOnlyAccessArray,   128,    portMPU_REGION_PRIVILEGED_READ_
↪WRITE }
 }

 &xTaskBuffer; // Holds the task's data structure.
};

int main( void )
{
TaskHandle_t xHandle;

 // Create a task from the const structure defined above.  The task handle
 // is requested (the second parameter is not NULL) but in this case just for
 // demonstration purposes as its not actually used.
 xTaskCreateRestricted( &xRegTest1Parameters, &xHandle );

 // Start the scheduler.
 vTaskStartScheduler();

 // Will only get here if there was insufficient memory to create the idle
 // and/or timer task.
 for( ;; );
}
```

Memory regions are assigned to a restricted task when the task is created by a call to xTaskCreateRestricted(). These regions can be redefined using vTaskAllocateMPURegions().

Example usage:

```
// Define an array of MemoryRegion_t structures that configures an MPU region
// allowing read/write access for 1024 bytes starting at the beginning of the
// ucOneKByte array.  The other two of the maximum 3 definable regions are
// unused so set to zero.
static const MemoryRegion_t xAltRegions[ portNUM_CONFIGURABLE_REGIONS ] =
{
 // Base address     Length      Parameters
 { ucOneKByte,       1024,       portMPU_REGION_READ_WRITE },
 { 0,                0,          0 },
 { 0,                0,          0 }
```

(continues on next page)

(continued from previous page)

```
};

void vATask( void *pvParameters )
{
 // This task was created such that it has access to certain regions of
 // memory as defined by the MPU configuration.  At some point it is
 // desired that these MPU regions are replaced with that defined in the
 // xAltRegions const struct above.  Use a call to vTaskAllocateMPURegions()
 // for this purpose.  NULL is used as the task handle to indicate that this
 // function should modify the MPU regions of the calling task.
 vTaskAllocateMPURegions( NULL, xAltRegions );

 // Now the task can continue its function, but from this point on can only
 // access its stack and the ucOneKByte array (unless any other statically
 // defined or shared regions have been declared elsewhere).
}
```

Parameters

- **pxTaskDefinition** –Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions. If configSUPPORT_STATIC_ALLOCATION is set to 1 the structure contains an additional member, which is used to point to a variable of type StaticTask_t - which is then used to hold the task's data structure.
- **pxCreatedTask** –Used to pass back a handle by which the created task can be referenced.
- **xTask** –The handle of the task being updated.
- **pxRegions** –A pointer to an MemoryRegion_t structure that contains the new memory region definitions.

void **vTaskDelete**(*TaskHandle_t* xTaskToDelete)

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

Remove a task from the RTOS real time kernel's management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death.c for sample code that utilises vTaskDelete ().

Example usage:

```
void vOtherFunction( void )
{
TaskHandle_t xHandle;

  // Create the task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle␣
↪);

  // Use the handle to delete the task.
  vTaskDelete( xHandle );
}
```

Parameters **xTaskToDelete** –The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

void **vTaskDelay**(const TickType_t xTicksToDelay)

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See xTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Example usage:

```
void vTaskFunction( void * pvParameters )
{
// Block for 500ms.
const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

  for( ;; )
  {
      // Simply toggle the LED every 500ms, blocking between each toggle.
      vToggleLED();
      vTaskDelay( xDelay );
  }
}
```

Parameters **xTicksToDelay** –The amount of time, in tick periods, that the calling task should block.

BaseType_t **xTaskDelayUntil**(TickType_t *const pxPreviousWakeTime, const TickType_t xTimeIncrement)

INCLUDE_xTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.

This function differs from vTaskDelay () in one important aspect: vTaskDelay () will cause a task to block for the specified number of ticks from the time vTaskDelay () is called. It is therefore difficult to use vTaskDelay () by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTaskDelay () may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay () specifies a wake time relative to the time at which the function is called, xTaskDelayUntil () specifies the absolute (exact) time at which it wishes to unblock.

The macro pdMS_TO_TICKS() can be used to calculate the number of ticks from a time specified in milliseconds with a resolution of one tick period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xFrequency = 10;
BaseType_t xWasDelayed;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount ();
    for( ;; )
    {
        // Wait for the next cycle.
        xWasDelayed = xTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here. xWasDelayed value can be used to determine
        // whether a deadline was missed if the code here took too long.
    }
}
```

Parameters

- **pxPreviousWakeTime** –Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within xTaskDelayUntil ().
- **xTimeIncrement** –The cycle time period. The task will be unblocked at time *pxPreviousWakeTime + xTimeIncrement. Calling xTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interface period.

Returns Value which can be used to check whether the task was actually delayed. Will be pdTRUE if the task way delayed and pdFALSE otherwise. A task will not be delayed if the next expected wake time is in the past.

BaseType_t **xTaskAbortDelay**(*TaskHandle_t* xTask)

INCLUDE_xTaskAbortDelay must be defined as 1 in FreeRTOSConfig.h for this function to be available.

A task will enter the Blocked state when it is waiting for an event. The event it is waiting for can be a temporal event (waiting for a time), such as when vTaskDelay() is called, or an event on an object, such as when xQueueReceive() or ulTaskNotifyTake() is called. If the handle of a task that is in the Blocked state is used in a call to xTaskAbortDelay() then the task will leave the Blocked state, and return from whichever function call placed the task into the Blocked state.

There is no 'FromISR' version of this function as an interrupt would need to know which object a task was blocked on in order to know which actions to take. For example, if the task was blocked on a queue the interrupt handler would then need to know if the queue was locked.

Parameters **xTask** –The handle of the task to remove from the Blocked state.

Returns If the task referenced by xTask was not in the Blocked state then pdFAIL is returned. Otherwise pdPASS is returned.

UBaseType_t **uxTaskPriorityGet**(const *TaskHandle_t* xTask)

INCLUDE_uxTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the priority of any task.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;
```

(continues on next page)

(continued from previous page)

```
  // Create a task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle␣
↪);

  // ...

  // Use the handle to obtain the priority of the created task.
  // It was created with tskIDLE_PRIORITY, but may have changed
  // it itself.
  if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
  {
      // The task has changed it's priority.
  }

  // ...

  // Is our priority higher than the created task?
  if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
  {
      // Our priority (obtained using NULL handle) is higher.
  }
}
```

Parameters **xTask** –Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns The priority of xTask.

UBaseType_t **uxTaskPriorityGetFromISR**(const *TaskHandle_t* xTask)

A version of uxTaskPriorityGet() that can be used from an ISR.

eTaskState **eTaskGetState**(*TaskHandle_t* xTask)

INCLUDE_eTaskGetState must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the state of any task. States are encoded by the eTaskState enumerated type.

Parameters **xTask** –Handle of the task to be queried.

Returns The state of xTask at the time the function was called. Note the state of the task might change between the function being called, and the functions return value being tested by the calling task.

void **vTaskGetInfo**(*TaskHandle_t* xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace, *eTaskState* eState)

configUSE_TRACE_FACILITY must be defined as 1 for this function to be available. See the configuration section for more information.

Populates a TaskStatus_t structure with information about a task.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;
TaskStatus_t xTaskDetails;

 // Obtain the handle of a task from its name.
 xHandle = xTaskGetHandle( "Task_Name" );
```

(continues on next page)

(continued from previous page)

```
 // Check the handle is not NULL.
 configASSERT( xHandle );

 // Use the handle to obtain further information about the task.
 vTaskGetInfo( xHandle,
               &xTaskDetails,
               pdTRUE, // Include the high water mark in xTaskDetails.
               eInvalid ); // Include the task state in xTaskDetails.
}
```

Parameters

- **xTask** –Handle of the task being queried. If xTask is NULL then information will be returned about the calling task.
- **pxTaskStatus** –A pointer to the TaskStatus_t structure that will be filled with information about the task referenced by the handle passed using the xTask parameter.
- **xGetFreeStackSpace** –The TaskStatus_t structure contains a member to report the stack high water mark of the task being queried. Calculating the stack high water mark takes a relatively long time, and can make the system temporarily unresponsive - so the xGetFreeStackSpace parameter is provided to allow the high water mark checking to be skipped. The high watermark value will only be written to the TaskStatus_t structure if xGetFreeStackSpace is not set to pdFALSE;
- **eState** –The TaskStatus_t structure contains a member to report the state of the task being queried. Obtaining the task state is not as fast as a simple assignment - so the eState parameter is provided to allow the state information to be omitted from the TaskStatus_t structure. To obtain state information then set eState to eInvalid - otherwise the value passed in eState will be reported as the task state in the TaskStatus_t structure.

void **vTaskPrioritySet**(*TaskHandle_t* xTask, UBaseType_t uxNewPriority)

INCLUDE_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.

Set the priority of any task.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

  // Create a task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle␣
↪);

  // ...

  // Use the handle to raise the priority of the created task.
  vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

  // ...

  // Use a NULL handle to raise our priority to the same value.
  vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```

Parameters

- **xTask** –Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- **uxNewPriority** –The priority to which the task will be set.

void **vTaskSuspend**(*TaskHandle_t* xTaskToSuspend)

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

  // Create a task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle␣
↪);

  // ...

  // Use the handle to suspend the created task.
  vTaskSuspend( xHandle );

  // ...

  // The created task will not run during this period, unless
  // another task calls vTaskResume( xHandle ).

  //...


  // Suspend ourselves.
  vTaskSuspend( NULL );

  // We cannot get here unless another task calls vTaskResume
  // with our handle as the parameter.
}
```

Parameters **xTaskToSuspend** –Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

void **vTaskResume**(*TaskHandle_t* xTaskToResume)

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Resumes a suspended task.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

  // Create a task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle␣
↪);

  // ...

  // Use the handle to suspend the created task.
  vTaskSuspend( xHandle );

  // ...

  // The created task will not run during this period, unless
  // another task calls vTaskResume( xHandle ).

  //...


  // Resume the suspended task ourselves.
  vTaskResume( xHandle );

  // The created task will once again get microcontroller processing
  // time in accordance with its priority within the system.
}
```

Parameters **xTaskToResume** –Handle to the task being readied.

BaseType_t **xTaskResumeFromISR**(*TaskHandle_t* xTaskToResume)

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

An implementation of vTaskResume() that can be called from within an ISR.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to xTaskResumeFromISR ().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Parameters **xTaskToResume** –Handle to the task being readied.

Returns pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

void **vTaskStartScheduler**(void)

Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when.

NOTE: In ESP-IDF the scheduler is started automatically during application startup, vTaskStartScheduler() should not be called from ESP-IDF applications.

See the demo application file main.c for an example of creating tasks and starting the kernel.

Example usage:

```
void vAFunction( void )
{
  // Create at least one task before starting the kernel.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );
```

(continues on next page)

(continued from previous page)

```
  // Start the real time kernel with preemption.
  vTaskStartScheduler ();

  // Will not get here unless a task calls vTaskEndScheduler ()
}
```

void **vTaskEndScheduler** (void)

NOTE: At the time of writing only the x86 real mode port, which runs on a PC in place of DOS, implements this function.

Stops the real time kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler () was called, as if vTaskStartScheduler () had just returned.

See the demo application file main. c in the demo/PC directory for an example that uses vTaskEndScheduler ().

vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the kernel to be freed - but will not free resources allocated by application tasks.

Example usage:

```
void vTaskCode( void * pvParameters )
{
  for( ;; )
  {
      // Task code goes here.

      // At some point we want to end the real time kernel processing
      // so call ...
      vTaskEndScheduler ();
  }
}

void vAFunction( void )
{
  // Create at least one task before starting the kernel.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

  // Start the real time kernel with preemption.
  vTaskStartScheduler ();

  // Will only get here when the vTaskCode () task has called
  // vTaskEndScheduler ().  When we get here we are back to single task
  // execution.
}
```

void **vTaskSuspendAll** (void)

Suspends the scheduler without disabling interrupts. Context switches will not occur while the scheduler is suspended.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

Example usage:

```
void vTask1( void * pvParameters )
{
  for( ;; )
  {
      // Task code goes here.

      // ...

      // At some point the task wants to perform a long operation during
      // which it does not want to get swapped out.  It cannot use
      // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
      // operation may cause interrupts to be missed - including the
      // ticks.

      // Prevent the real time kernel swapping out the task.
      vTaskSuspendAll ();

      // Perform the operation here.  There is no need to use critical
      // sections as we have all the microcontroller processing time.
      // During this time interrupts will still operate and the kernel
      // tick count will be maintained.

      // ...

      // The operation is complete.  Restart the kernel.
      xTaskResumeAll ();
  }
}
```

BaseType_t **xTaskResumeAll** (void)

Resumes scheduler activity after it was suspended by a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Example usage:

```
void vTask1( void * pvParameters )
{
  for( ;; )
  {
      // Task code goes here.

      // ...

      // At some point the task wants to perform a long operation during
      // which it does not want to get swapped out.  It cannot use
      // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
      // operation may cause interrupts to be missed - including the
      // ticks.

      // Prevent the real time kernel swapping out the task.
      vTaskSuspendAll ();

      // Perform the operation here.  There is no need to use critical
      // sections as we have all the microcontroller processing time.
      // During this time interrupts will still operate and the real
      // time kernel tick count will be maintained.

      // ...
```

(continues on next page)

(continued from previous page)

```
        // The operation is complete.  Restart the kernel.  We want to force
        // a context switch - but there is no point if resuming the scheduler
        // caused a context switch already.
        if( !xTaskResumeAll () )
        {
             taskYIELD ();
        }
    }
}
```

Returns If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

TickType_t **xTaskGetTickCount**(void)

Returns The count of ticks since vTaskStartScheduler was called.

TickType_t **xTaskGetTickCountFromISR**(void)

This is a version of xTaskGetTickCount() that is safe to be called from an ISR - provided that TickType_t is the natural word size of the microcontroller being used or interrupt nesting is either not supported or not being used.

Returns The count of ticks since vTaskStartScheduler was called.

UBaseType_t **uxTaskGetNumberOfTasks**(void)

Returns The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

char ***pcTaskGetName**(*TaskHandle_t* xTaskToQuery)

Returns The text (human readable) name of the task referenced by the handle xTaskToQuery. A task can query its own name by either passing in its own handle, or by setting xTaskToQuery to NULL.

TaskHandle_t **xTaskGetHandle**(const char *pcNameToQuery)

NOTE: This function takes a relatively long time to complete and should be used sparingly.

Returns The handle of the task that has the human readable name pcNameToQuery. NULL is returned if no matching name is found. INCLUDE_xTaskGetHandle must be set to 1 in FreeRTOSConfig.h for pcTaskGetHandle() to be available.

UBaseType_t **uxTaskGetStackHighWaterMark**(*TaskHandle_t* xTask)

Returns the high water mark of the stack associated with xTask.

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in bytes not words, unlike vanilla FreeRTOS) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

uxTaskGetStackHighWaterMark() and uxTaskGetStackHighWaterMark2() are the same except for their return type. Using configSTACK_DEPTH_TYPE allows the user to determine the return type. It gets around the problem of the value overflowing on 8-bit types without breaking backward compatibility for applications that expect an 8-bit return type.

Parameters **xTask** –Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

Returns The smallest amount of free stack space there has been (in bytes not words, unlike vanilla FreeRTOS) since the task referenced by xTask was created.

configSTACK_DEPTH_TYPE **uxTaskGetStackHighWaterMark2** (*TaskHandle_t* xTask)

Returns the start of the stack associated with xTask.

INCLUDE_uxTaskGetStackHighWaterMark2 must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in words, so on a 32 bit machine a value of 1 means 4 bytes) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

uxTaskGetStackHighWaterMark() and uxTaskGetStackHighWaterMark2() are the same except for their return type. Using configSTACK_DEPTH_TYPE allows the user to determine the return type. It gets around the problem of the value overflowing on 8-bit types without breaking backward compatibility for applications that expect an 8-bit return type.

Parameters **xTask** –Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

Returns The smallest amount of free stack space there has been (in words, so actual spaces on the stack rather than bytes) since the task referenced by xTask was created.

uint8_t ***pxTaskGetStackStart** (*TaskHandle_t* xTask)

Returns the start of the stack associated with xTask.

INCLUDE_pxTaskGetStackStart must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the lowest stack memory address, regardless of whether the stack grows up or down.

Parameters **xTask** –Handle of the task associated with the stack returned. Set xTask to NULL to return the stack of the calling task.

Returns A pointer to the start of the stack.

void **vTaskSetApplicationTaskTag** (*TaskHandle_t* xTask, *TaskHookFunction_t* pxHookFunction)

Sets pxHookFunction to be the task hook function used by the task xTask.

Parameters

- **xTask** –Handle of the task to set the hook function for Passing xTask as NULL has the effect of setting the calling tasks hook function.
- **pxHookFunction** –Pointer to the hook function.

TaskHookFunction_t **xTaskGetApplicationTaskTag** (*TaskHandle_t* xTask)

Returns the pxHookFunction value assigned to the task xTask. Do not call from an interrupt service routine - call xTaskGetApplicationTaskTagFromISR() instead.

TaskHookFunction_t **xTaskGetApplicationTaskTagFromISR** (*TaskHandle_t* xTask)

Returns the pxHookFunction value assigned to the task xTask. Can be called from an interrupt service routine.

void **vTaskSetThreadLocalStoragePointer** (*TaskHandle_t* xTaskToSet, BaseType_t xIndex, void *pvValue)

Set local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the configNUM_THREAD_LOCAL_STORAGE_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Parameters

- **xTaskToSet** –Task to set thread local storage pointer for
- **xIndex** –The index of the pointer to set, from 0 to configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1.
- **pvValue** –Pointer value to set.

void ***pvTaskGetThreadLocalStoragePointer** (*TaskHandle_t* xTaskToQuery, BaseType_t xIndex)

Get local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the configNUM_THREAD_LOCAL_STORAGE_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Parameters
- **xTaskToQuery** –Task to get thread local storage pointer for
- **xIndex** –The index of the pointer to get, from 0 to configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1.

Returns Pointer value

void **vTaskSetThreadLocalStoragePointerAndDelCallback**(*TaskHandle_t* xTaskToSet, BaseType_t xIndex, void *pvValue, *TlsDeleteCallbackFunction_t* pvDelCallback)

Set local storage pointer and deletion callback.

Each task contains an array of pointers that is dimensioned by the configNUM_THREAD_LOCAL_STORAGE_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Local storage pointers set for a task can reference dynamically allocated resources. This function is similar to vTaskSetThreadLocalStoragePointer, but provides a way to release these resources when the task gets deleted. For each pointer, a callback function can be set. This function will be called when task is deleted, with the local storage pointer index and value as arguments.

Parameters
- **xTaskToSet** –Task to set thread local storage pointer for
- **xIndex** –The index of the pointer to set, from 0 to configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1.
- **pvValue** –Pointer value to set.
- **pvDelCallback** –Function to call to dispose of the local storage pointer when the task is deleted.

void **vApplicationGetIdleTaskMemory**(StaticTask_t **ppxIdleTaskTCBBuffer, StackType_t **ppxIdleTaskStackBuffer, uint32_t *pulIdleTaskStackSize)

This function is used to provide a statically allocated block of memory to FreeRTOS to hold the Idle Task TCB. This function is required when configSUPPORT_STATIC_ALLOCATION is set. For more information see this URI: https://www.FreeRTOS.org/a00110.html#configSUPPORT_STATIC_ALLOCATION

Parameters
- **ppxIdleTaskTCBBuffer** –A handle to a statically allocated TCB buffer
- **ppxIdleTaskStackBuffer** –A handle to a statically allocated Stack buffer for thie idle task
- **pulIdleTaskStackSize** –A pointer to the number of elements that will fit in the allocated stack buffer

BaseType_t **xTaskCallApplicationTaskHook**(*TaskHandle_t* xTask, void *pvParameter)

Calls the hook function associated with xTask. Passing xTask as NULL has the effect of calling the Running tasks (the calling task) hook function.

Parameters
- **xTask** –Handle of the task to call the hook for.
- **pvParameter** –Parameter passed to the hook function for the task to interpret as it wants. The return value is the value returned by the task hook function registered by the user.

TaskHandle_t **xTaskGetIdleTaskHandle**(void)

xTaskGetIdleTaskHandle() is only available if INCLUDE_xTaskGetIdleTaskHandle is set to 1 in FreeRTOSConfig.h.

Simply returns the handle of the idle task. It is not valid to call xTaskGetIdleTaskHandle() before the scheduler has been started.

UBaseType_t **uxTaskGetSystemState** (TaskStatus_t *const pxTaskStatusArray, const UBaseType_t uxArraySize, uint32_t *const pulTotalRunTime)

configUSE_TRACE_FACILITY must be defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.

uxTaskGetSystemState() populates an TaskStatus_t structure for each task in the system. TaskStatus_t structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task. See the TaskStatus_t structure definition in this file for the full member list.

NOTE: This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Example usage:

```
// This example demonstrates how a human readable table of run time stats
// information is generated from raw data provided by uxTaskGetSystemState().
// The human readable table is written to pcWriteBuffer
void vTaskGetRunTimeStats( char *pcWriteBuffer )
{
TaskStatus_t *pxTaskStatusArray;
volatile UBaseType_t uxArraySize, x;
uint32_t ulTotalRunTime, ulStatsAsPercentage;

    // Make sure the write buffer does not contain a string.
*pcWriteBuffer = 0x00;

    // Take a snapshot of the number of tasks in case it changes while this
    // function is executing.
    uxArraySize = uxTaskGetNumberOfTasks();

    // Allocate a TaskStatus_t structure for each task.  An array could be
    // allocated statically at compile time.
    pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

    if( pxTaskStatusArray != NULL )
    {
        // Generate raw status information about each task.
        uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &
↪ulTotalRunTime );

        // For percentage calculations.
        ulTotalRunTime /= 100UL;

        // Avoid divide by zero errors.
        if( ulTotalRunTime > 0 )
        {
            // For each populated position in the pxTaskStatusArray array,
            // format the raw data as human readable ASCII data
            for( x = 0; x < uxArraySize; x++ )
            {
                // What percentage of the total run time has the task used?
                // This will always be rounded down to the nearest integer.
                // ulTotalRunTimeDiv100 has already been divided by 100.
                ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter␣
↪/ ulTotalRunTime;

                if( ulStatsAsPercentage > 0UL )
                {
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",␣
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter,␣
↪ulStatsAsPercentage );
```

(continues on next page)

(continued from previous page)

```
                }
                else
                {
                    // If the percentage is zero here then the task has
                    // consumed less than 1% of the total run time.
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",␣
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter );
                }

                pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
            }
        }

        // The array is no longer needed, free the memory it consumes.
        vPortFree( pxTaskStatusArray );
    }
}
```

Parameters

- **pxTaskStatusArray** –A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
- **uxArraySize** –The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array, or the number of TaskStatus_t structures contained in the array, not by the number of bytes in the array.
- **pulTotalRunTime** –If configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h then *pulTotalRunTime is set by uxTaskGetSystemState() to the total run time (as defined by the run time stats clock, see https://www.FreeRTOS.org/rtos-run-time-stats.html) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time information.

Returns The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

void **vTaskList**(char *pcWriteBuffer)

List all the current tasks.

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Lists all the current tasks, along with their current state and stack usage high water mark.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

PLEASE NOTE:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskList() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task names, states and stack usage.

vTaskList() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskList().

Parameters **pcWriteBuffer** –A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

void **vTaskGetRunTimeStats**(char *pcWriteBuffer)

Get the state of running tasks as a string

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Setting configGENERATE_RUN_TIME_STATS to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro. Calling vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

NOTE 2:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

vTaskGetRunTimeStats() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskGetRunTimeStats().

Parameters **pcWriteBuffer** –A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

uint32_t **ulTaskGetIdleRunTimeCounter**(void)

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

Setting configGENERATE_RUN_TIME_STATS to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro. While uxTaskGetSystemState() and vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, ulTaskGetIdleRunTimeCounter() returns the total execution time of just the idle task.

Returns The total run time of the idle task. This is the amount of time the idle task has actually been executing. The unit of time is dependent on the frequency configured using the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() macros.

BaseType_t **xTaskGenericNotify**(*TaskHandle_t* xTaskToNotify, UBaseType_t uxIndexToNotify, uint32_t ulValue, *eNotifyAction* eAction, uint32_t *pulPreviousNotificationValue)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Sends a direct to task notification to a task, with an optional value and action.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotify() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotify() is equivalent to calling xTaskNotifyIndexed() with the uxIndexToNotify parameter set to 0.

eSetBits - The target notification value is bitwise ORed with ulValue. xTaskNotifyIndexed() always returns pdPASS in this case.

eIncrement - The target notification value is incremented. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

eSetValueWithOverwrite - The target notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification at the same array index (the task already had a notification pending at that index). xTaskNotifyIndexed() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending at the same array index then the target notification value is set to ulValue and xTaskNotifyIndexed() will return pdPASS. If the task being notified already had a notification pending at the same array index then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification at the specified array index without the notification value at that index being updated. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

Parameters

- **xTaskToNotify** –The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().

- **uxIndexToNotify** –The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotify() does not have this parameter and always sends notifications to index 0.
- **ulValue** –Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter.
- **eAction** –Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows:
- **pulPreviousNotificationValue** –- Can be used to pass out the subject task's notification value before any bits are modified by the notify function.

Returns Dependent on the value of eAction. See the description of the eAction parameter.

BaseType_t **xTaskGenericNotifyFromISR**(*TaskHandle_t* xTaskToNotify, UBaseType_t uxIndexToNotify, uint32_t ulValue, *eNotifyAction* eAction, uint32_t *pulPreviousNotificationValue, BaseType_t *pxHigherPriorityTaskWoken)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

A version of xTaskNotifyIndexed() that can be used from an interrupt service routine (ISR).

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyFromISR() is equivalent to calling xTaskNotifyIndexedFromISR() with the uxIndexToNotify parameter set to 0.

eSetBits - The task's notification value is bitwise ORed with ulValue. xTaskNotify() always returns pdPASS in this case.

eIncrement - The task's notification value is incremented. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

eSetValueWithOverwrite - The task's notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). xTaskNotify() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending then the task's notification value is set to ulValue and xTaskNotify() will return pdPASS. If the task being notified already had a notification pending then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification without its notification value being updated. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

Parameters

- **`uxIndexToNotify`** –The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyFromISR() does not have this parameter and always sends notifications to index 0.
- **`xTaskToNotify`** –The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
- **`ulValue`** –Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter.
- **`eAction`** –Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows:
- **`pulPreviousNotificationValue`** –- Can be used to pass out the subject task's notification value before any bits are modified by the notify function.
- **`pxHigherPriorityTaskWoken`** –xTaskNotifyFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If xTaskNotifyFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

Returns Dependent on the value of eAction. See the description of the eAction parameter.

BaseType_t **`xTaskGenericNotifyWait`**(UBaseType_t uxIndexToWaitOn, uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait)

Waits for a direct to task notification to be pending at a given index within an array of direct to task notifications.

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyWait() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotifyWait() is equivalent to calling xTaskNotifyWaitIndexed() with the uxIndexToWaitOn parameter set to 0.

Parameters

- **`uxIndexToWaitOn`** –The index within the calling task's array of notification values on which the calling task will wait for a notification to be received. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyWait() does not have this parameter and always waits for notifications on index 0.
- **`ulBitsToClearOnEntry`** –Bits that are set in ulBitsToClearOnEntry value will be cleared in the calling task's notification value before the task is marked as waiting for a new notification (provided a notification is not already pending). Optionally blocks if no notifications are pending. Setting ulBitsToClearOnEntry to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0. Setting ulBitsToClearOnEntry to 0 will leave the task's notification value unchanged.
- **`ulBitsToClearOnExit`** –If a notification is pending or received before the calling task exits the xTaskNotifyWait() function then the task's notification value (see the xTaskNotify() API function) is passed out using the pulNotificationValue parameter. Then any bits that are set in ulBitsToClearOnExit will be cleared in the task's notification value (note *pulNotificationValue is set before any bits are cleared). Setting ulBitsToClearOnExit to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0 before the function exits. Setting ulBitsToClearOnExit to 0 will leave the task's notification value unchanged when the function exits (in which case the value passed out in pulNotificationValue will match the task's notification value).
- **`pulNotificationValue`** –Used to pass the task's notification value out of the function. Note the value passed out will not be effected by the clearing of any bits caused by ulBitsToClearOnExit being non-zero.
- **`xTicksToWait`** –The maximum amount of time that the task should wait in the Blocked state for a notification to be received, should a notification not already be pending when xTaskNotifyWait() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICKS(value_in_ms) can be used to convert a time specified in milliseconds to a time specified in ticks.

Returns If a notification was received (including notifications that were already pending when xTaskNotifyWait was called) then pdPASS is returned. Otherwise pdFAIL is returned.

void **`vTaskGenericNotifyGiveFromISR`**(*TaskHandle_t* xTaskToNotify, UBaseType_t uxIndexToNotify, BaseType_t *pxHigherPriorityTaskWoken)

A version of xTaskNotifyGiveIndexed() that can be called from an interrupt service routine (ISR).

See https://www.FreeRTOS.org/RTOS-task-notifications.html for more details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

vTaskNotifyGiveIndexedFromISR() is intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given from an ISR using the xSemaphoreGiveFromISR() API function, the equivalent action that instead uses a task notification is vTaskNotifyGiveIndexedFromISR().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTakeIndexed() API function rather than the xTaskNotifyWaitIndexed() API function.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyGiveFromISR() is equivalent to calling xTaskNotifyGiveIndexedFromISR() with the uxIndexToNotify parameter set to 0.

Parameters

- **`xTaskToNotify`** –The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
- **`uxIndexToNotify`** –The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyGiveFromISR() does not have this parameter and always sends notifications to index 0.
- **`pxHigherPriorityTaskWoken`** –vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

uint32_t **`ulTaskGenericNotifyTake`**(UBaseType_t uxIndexToWaitOn, BaseType_t xClearCountOnExit, TickType_t xTicksToWait)

Waits for a direct to task notification on a particular index in the calling task's notification array in a manner similar to taking a counting semaphore.

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

ulTaskNotifyTakeIndexed() is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. Actual FreeRTOS semaphores are taken using the xSemaphoreTake() API function, the equivalent action that instead uses a task notification is ulTaskNotifyTakeIndexed().

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the xTaskNotifyGiveIndexed() macro, or xTaskNotifyIndex() function with the eAction parameter set to eIncrement.

ulTaskNotifyTakeIndexed() can either clear the task's notification value at the array index specified by the uxIndexToWaitOn parameter to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use ulTaskNotifyTakeIndexed() to [optionally] block to wait for the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

Where as xTaskNotifyWaitIndexed() will return when a notification is pending, ulTaskNotifyTakeIndexed() will return when the task's notification value is not zero.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyTake() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling ulTaskNotifyTake() is equivalent to calling ulTaskNotifyTakeIndexed() with the uxIndexToWaitOn parameter set to 0.

Parameters

- **`uxIndexToWaitOn`** –The index within the calling task's array of notification values on which the calling task will wait for a notification to be non-zero. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyTake() does not have this parameter and always waits for notifications on index 0.
- **`xClearCountOnExit`** –if xClearCountOnExit is pdFALSE then the task's notification value is decremented when the function exits. In this way the notification value acts like a counting semaphore. If xClearCountOnExit is not pdFALSE then the task's notification value is cleared to zero when the function exits. In this way the notification value acts like a binary semaphore.
- **`xTicksToWait`** –The maximum amount of time that the task should wait in the Blocked state for the task's notification value to be greater than zero, should the count not already be greater than zero when ulTaskNotifyTake() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICKS(value_in_ms) can be used to convert a time specified in milliseconds to a time specified in ticks.

Returns The task's notification count before it is either cleared to zero or decremented (see the xClearCountOnExit parameter).

BaseType_t **`xTaskGenericNotifyStateClear`**(*TaskHandle_t* xTask, UBaseType_t uxIndexToClear)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

If a notification is sent to an index within the array of notifications then the notification at that index is said to be 'pending' until it is read or explicitly cleared by the receiving task. xTaskNotifyStateClearIndexed() is the function that clears a pending notification without reading the notification value. The notification value at the same array index is not altered. Set xTask to NULL to clear the notification state of the calling task.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyStateClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyStateClear() is equivalent to calling xTaskNotifyStateClearIndexed() with the uxIndexToNotify parameter set to 0.

Parameters

- **xTask** –The handle of the RTOS task that will have a notification state cleared. Set xTask to NULL to clear a notification state in the calling task. To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().
- **uxIndexToClear** –The index within the target task's array of notification values to act upon. For example, setting uxIndexToClear to 1 will clear the state of the notification at index 1 within the array. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyStateClear() does not have this parameter and always acts on the notification at index 0.

Returns pdTRUE if the task's notification state was set to eNotWaitingNotification, otherwise pdFALSE.

uint32_t **ulTaskGenericNotifyValueClear**(*TaskHandle_t* xTask, UBaseType_t uxIndexToClear, uint32_t ulBitsToClear)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

ulTaskNotifyValueClearIndexed() clears the bits specified by the ulBitsToClear bit mask in the notification value at array index uxIndexToClear of the task referenced by xTask.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyValueClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling ulTaskNotifyValueClear() is equivalent to calling ulTaskNotifyValueClearIndexed() with the uxIndexToClear parameter set to 0.

Parameters

- **xTask** –The handle of the RTOS task that will have bits in one of its notification values cleared. Set xTask to NULL to clear bits in a notification value of the calling task. To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().
- **uxIndexToClear** –The index within the target task's array of notification values in which to clear the bits. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyValueClear() does not have this parameter and always clears bits in the notification value at index 0.
- **ulBitsToClear** –Bit mask of the bits to clear in the notification value of xTask. Set a bit to 1 to clear the corresponding bits in the task's notification value. Set ulBitsToClear to 0xffffffff (UINT_MAX on 32-bit architectures) to clear the notification value to 0. Set ulBitsToClear to 0 to query the task's notification value without clearing any bits.

Returns The value of the target task's notification value before the bits specified by ulBitsToClear were cleared.

void **vTaskSetTimeOutState**(TimeOut_t *const pxTimeOut)

BaseType_t **xTaskCheckForTimeOut**(TimeOut_t *const pxTimeOut, TickType_t *const pxTicksToWait)

Determines if pxTicksToWait ticks has passed since a time was captured using a call to vTaskSetTimeOutState(). The captured time includes the tick count and the number of times the tick count has overflowed.

Example Usage:

```
// Driver library function used to receive uxWantedBytes from an Rx buffer
// that is filled by a UART interrupt. If there are not enough bytes in the
// Rx buffer then the task enters the Blocked state until it is notified that
// more data has been placed into the buffer. If there is still not enough
// data then the task re-enters the Blocked state, and xTaskCheckForTimeOut()
// is used to re-calculate the Block time to ensure the total amount of time
// spent in the Blocked state does not exceed MAX_TIME_TO_WAIT. This
// continues until either the buffer contains at least uxWantedBytes bytes,
// or the total amount of time spent in the Blocked state reaches
// MAX_TIME_TO_WAIT – at which point the task reads however many bytes are
// available up to a maximum of uxWantedBytes.

size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
size_t uxReceived = 0;
TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
TimeOut_t xTimeOut;

    // Initialize xTimeOut.  This records the time at which this function
    // was entered.
    vTaskSetTimeOutState( &xTimeOut );

    // Loop until the buffer contains the wanted number of bytes, or a
    // timeout occurs.
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        // The buffer didn't contain enough data so this task is going to
        // enter the Blocked state. Adjusting xTicksToWait to account for
        // any time that has been spent in the Blocked state within this
        // function so far to ensure the total amount of time spent in the
        // Blocked state does not exceed MAX_TIME_TO_WAIT.
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            //Timed out before the wanted number of bytes were available,
            // exit the loop.
            break;
        }

        // Wait for a maximum of xTicksToWait ticks to be notified that the
        // receive interrupt has placed more data into the buffer.
        ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }

    // Attempt to read uxWantedBytes from the receive buffer into pucBuffer.
    // The actual number of bytes read (which might be less than
    // uxWantedBytes) is returned.
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance,
                                                pucBuffer,
                                                uxWantedBytes );

    return uxReceived;
}
```

See also:

https://www.FreeRTOS.org/xTaskCheckForTimeOut.html

Parameters

- **`pxTimeOut`** –The time status as captured previously using vTaskSetTimeOutState. If the timeout has not yet occurred, it is updated to reflect the current time status.
- **`pxTicksToWait`** –The number of ticks to check for timeout i.e. if pxTicksToWait ticks have passed since pxTimeOut was last updated (either by vTaskSetTimeOutState()

or xTaskCheckForTimeOut()), the timeout has occurred. If the timeout has not occurred, pxTicksToWait is updated to reflect the number of remaining ticks.

Returns If timeout has occurred, pdTRUE is returned. Otherwise pdFALSE is returned and pxTicksToWait is updated to reflect the number of remaining ticks.

BaseType_t **xTaskCatchUpTicks**(TickType_t xTicksToCatchUp)

Macros

tskKERNEL_VERSION_NUMBER

tskKERNEL_VERSION_MAJOR

tskKERNEL_VERSION_MINOR

tskKERNEL_VERSION_BUILD

tskMPU_REGION_READ_ONLY

tskMPU_REGION_READ_WRITE

tskMPU_REGION_EXECUTE_NEVER

tskMPU_REGION_NORMAL_MEMORY

tskMPU_REGION_DEVICE_MEMORY

tskDEFAULT_INDEX_TO_NOTIFY

tskNO_AFFINITY

tskIDLE_PRIORITY

Defines the priority used by the idle task. This must not be modified.

taskYIELD()

Macro for forcing a context switch.

taskENTER_CRITICAL()

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

Note: This may alter the stack (depending on the portable implementation) so must be used with care!

---

taskENTER_CRITICAL_FROM_ISR()

taskENTER_CRITICAL_ISR()

taskEXIT_CRITICAL()

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

Note: This may alter the stack (depending on the portable implementation) so must be used with care!

taskEXIT_CRITICAL_FROM_ISR(x)

taskEXIT_CRITICAL_ISR()

taskDISABLE_INTERRUPTS()

Macro to disable all maskable interrupts.

taskENABLE_INTERRUPTS()

Macro to enable microcontroller interrupts.

taskSCHEDULER_SUSPENDED

taskSCHEDULER_NOT_STARTED

taskSCHEDULER_RUNNING

vTaskDelayUntil(pxPreviousWakeTime, xTimeIncrement)

xTaskNotify(xTaskToNotify, ulValue, eAction)

xTaskNotifyIndexed(xTaskToNotify, uxIndexToNotify, ulValue, eAction)

xTaskNotifyAndQuery(xTaskToNotify, ulValue, eAction, pulPreviousNotifyValue)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

xTaskNotifyAndQueryIndexed() performs the same operation as xTaskNotifyIndexed() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQuery() performs the same operation as xTaskNotify() with the addition that it also returns the subject task's prior notification value (the notification value as it was at the time the function is called, rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQueryIndexed(xTaskToNotify, uxIndexToNotify, ulValue, eAction, pulPreviousNotifyValue)

xTaskNotifyFromISR(xTaskToNotify, ulValue, eAction, pxHigherPriorityTaskWoken)

xTaskNotifyIndexedFromISR(xTaskToNotify, uxIndexToNotify, ulValue, eAction, pxHigherPriorityTaskWoken)

xTaskNotifyAndQueryIndexedFromISR(xTaskToNotify, uxIndexToNotify, ulValue, eAction, pulPreviousNotificationValue, pxHigherPriorityTaskWoken)

See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

xTaskNotifyAndQueryIndexedFromISR() performs the same operation as xTaskNotifyIndexedFromISR() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than at the time the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQueryFromISR() performs the same operation as xTaskNotifyFromISR() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than at the time the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQueryFromISR (xTaskToNotify, ulValue, eAction, pulPreviousNotificationValue, pxHigherPriorityTaskWoken)

xTaskNotifyWait (ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait)

xTaskNotifyWaitIndexed (uxIndexToWaitOn, ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait)

xTaskNotifyGiveIndexed (xTaskToNotify, uxIndexToNotify)

Sends a direct to task notification to a particular index in the target task's notification array in a manner similar to giving a counting semaphore.

See https://www.FreeRTOS.org/RTOS-task-notifications.html for more details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these macros to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

xTaskNotifyGiveIndexed() is a helper macro intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the xSemaphoreGive() API function, the equivalent action that instead uses a task notification is xTaskNotifyGiveIndexed().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTakeIndexed() API function rather than the xTaskNotifyWaitIndexed() API function.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyGive() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotifyGive() is equivalent to calling xTaskNotifyGiveIndexed() with the uxIndexToNotify parameter set to 0.

Parameters

- **xTaskToNotify** –The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
- **uxIndexToNotify** –The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyGive() does not have this parameter and always sends notifications to index 0.

Returns xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement - so pdPASS is always returned.

xTaskNotifyGive (xTaskToNotify)

vTaskNotifyGiveFromISR (xTaskToNotify, pxHigherPriorityTaskWoken)

vTaskNotifyGiveIndexedFromISR (xTaskToNotify, uxIndexToNotify, pxHigherPriorityTaskWoken)

ulTaskNotifyTake(xClearCountOnExit, xTicksToWait)

ulTaskNotifyTakeIndexed(uxIndexToWaitOn, xClearCountOnExit, xTicksToWait)

xTaskNotifyStateClear(xTask)

xTaskNotifyStateClearIndexed(xTask, uxIndexToClear)

ulTaskNotifyValueClear(xTask, ulBitsToClear)

ulTaskNotifyValueClearIndexed(xTask, uxIndexToClear, ulBitsToClear)

Type Definitions

typedef struct tskTaskControlBlock ***TaskHandle_t**

typedef BaseType_t (***TaskHookFunction_t**)(void*)

typedef void (***TlsDeleteCallbackFunction_t**)(int, void*)

Prototype of local storage pointer deletion callback.

Enumerations

enum **eTaskState**

Task states returned by eTaskGetState.

Values:

enumerator **eRunning**

enumerator **eReady**

enumerator **eBlocked**

enumerator **eSuspended**

enumerator **eDeleted**

enumerator **eInvalid**

enum **eNotifyAction**

Values:

enumerator **eNoAction**

enumerator **eSetBits**

enumerator **eIncrement**

enumerator **eSetValueWithOverwrite**

enumerator **eSetValueWithoutOverwrite**

enum **eSleepModeStatus**

Possible return values for eTaskConfirmSleepModeStatus().

Values:

enumerator **eAbortSleep**

enumerator **eStandardSleep**

enumerator **eNoTasksWaitingTimeout**

Queue API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/queue.h

Functions

BaseType_t **xQueueGenericSend**(*QueueHandle_t* xQueue, const void *const pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition)

It is preferred that the macros xQueueSend(), xQueueSendToFront() and xQueueSendToBack() are used in place of calling this function directly.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

 // ...

 if( xQueue1 != 0 )
 {
```

(continues on next page)

(continued from previous page)

```
    // Send an uint32_t.  Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10,␣
↪queueSEND_TO_BACK ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0,␣
↪queueSEND_TO_BACK );
}

// ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **xTicksToWait** –The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.
- **xCopyPosition** –Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

Returns pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

BaseType_t **xQueuePeek**(*QueueHandle_t* xQueue, void *const pvBuffer, TickType_t xTicksToWait)

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueueReceive().

This macro must not be used in an interrupt service routine. See xQueuePeekFromISR() for an alternative that can be called from an interrupt service routine.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
```

(continues on next page)

(continued from previous page)

```
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
 if( xQueue == 0 )
 {
     // Failed to create the queue.
 }

 // ...

 // Send a pointer to a struct AMessage object.  Don't block if the
 // queue is already full.
 pxMessage = & xMessage;
 xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

 // ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

 if( xQueue != 0 )
 {
     // Peek a message on the created queue.  Block for 10 ticks if a
     // message is not immediately available.
     if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
     {
         // pcRxedMessage now points to the struct AMessage variable posted
         // by vATask, but the item still remains on the queue.
     }
 }

 // ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue from which the item is to be received.
- **pvBuffer** –Pointer to the buffer into which the received item will be copied.
- **xTicksToWait** –The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. xQueuePeek() will return immediately if xTicksToWait is 0 and the queue is empty.

Returns pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

BaseType_t **xQueuePeekFromISR**(*QueueHandle_t* xQueue, void *const pvBuffer)

A version of xQueuePeek() that can be called from an interrupt service routine (ISR).

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueueReceive().

Parameters

- **xQueue** –The handle to the queue from which the item is to be received.

- **pvBuffer** –Pointer to the buffer into which the received item will be copied.

Returns pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

BaseType_t **xQueueReceive**(*QueueHandle_t* xQueue, void *const pvBuffer, TickType_t xTicksToWait)

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items are removed from the queue.

This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
 if( xQueue == 0 )
 {
     // Failed to create the queue.
 }

 // ...

 // Send a pointer to a struct AMessage object.  Don't block if the
 // queue is already full.
 pxMessage = & xMessage;
 xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

 // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

 if( xQueue != 0 )
 {
     // Receive a message on the created queue.  Block for 10 ticks if a
     // message is not immediately available.
     if( xQueueReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
     {
         // pcRxedMessage now points to the struct AMessage variable posted
         // by vATask.
     }
 }
```

(continues on next page)

(continued from previous page)

```
 // ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue from which the item is to be received.
- **pvBuffer** –Pointer to the buffer into which the received item will be copied.
- **xTicksToWait** –The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. xQueueReceive() will return immediately if xTicksToWait is zero and the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

UBaseType_t **uxQueueMessagesWaiting**(const *QueueHandle_t* xQueue)

Return the number of messages stored in a queue.

Parameters **xQueue** –A handle to the queue being queried.
Returns The number of messages available in the queue.

UBaseType_t **uxQueueSpacesAvailable**(const *QueueHandle_t* xQueue)

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

Parameters **xQueue** –A handle to the queue being queried.
Returns The number of spaces available in the queue.

void **vQueueDelete**(*QueueHandle_t* xQueue)

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters **xQueue** –A handle to the queue to be deleted.

BaseType_t **xQueueGenericSendFromISR**(*QueueHandle_t* xQueue, const void *const pvItemToQueue, BaseType_t *const pxHigherPriorityTaskWoken, const BaseType_t xCopyPosition)

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() be used in place of calling this function directly. xQueueGiveFromISR() is an equivalent for use by semaphores that don't actually copy any data.

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWokenByPost;

 // We have not woken a task at the start of the ISR.
 xHigherPriorityTaskWokenByPost = pdFALSE;

 // Loop until the buffer is empty.
 do
 {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );
```

(continues on next page)

(continued from previous page)

```
    // Post each byte.
    xQueueGenericSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWokenByPost,
↪ queueSEND_TO_BACK );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.  Note that the
// name of the yield function required is port specific.
if( xHigherPriorityTaskWokenByPost )
{
    taskYIELD_YIELD_FROM_ISR();
}
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **pxHigherPriorityTaskWoken** –**[out]** xQueueGenericSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueGenericSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.
- **xCopyPosition** –Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

Returns pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

BaseType_t **xQueueGiveFromISR**(*QueueHandle_t* xQueue, BaseType_t *const pxHigherPriorityTaskWoken)

BaseType_t **xQueueReceiveFromISR**(*QueueHandle_t* xQueue, void *const pvBuffer, BaseType_t *const pxHigherPriorityTaskWoken)

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Example usage:

```
QueueHandle_t xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
char cValueToPost;
const TickType_t xTicksToWait = ( TickType_t )0xff;

 // Create a queue capable of containing 10 characters.
 xQueue = xQueueCreate( 10, sizeof( char ) );
 if( xQueue == 0 )
 {
     // Failed to create the queue.
 }

 // ...

 // Post some characters that will be used within an ISR.  If the queue
 // is full then this task will block for xTicksToWait ticks.
 cValueToPost = 'a';
 xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
```

(continues on next page)

(continued from previous page)

```
 cValueToPost = 'b';
 xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

 // ... keep posting characters ... this task may block when the queue
 // becomes full.

 cValueToPost = 'c';
 xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
}

// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
BaseType_t xTaskWokenByReceive = pdFALSE;
char cRxedChar;

 while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &
↪xTaskWokenByReceive) )
 {
     // A character was received.  Output the character now.
     vOutputCharacter( cRxedChar );

     // If removing the character from the queue woke the task that was
     // posting onto the queue cTaskWokenByReceive will have been set to
     // pdTRUE.  No matter how many times this loop iterates only one
     // task will be woken.
 }

 if( cTaskWokenByPost != ( char ) pdFALSE;
 {
     taskYIELD ();
 }
}
```

Parameters

- **xQueue** –The handle to the queue from which the item is to be received.
- **pvBuffer** –Pointer to the buffer into which the received item will be copied.
- **pxHigherPriorityTaskWoken** –**[out]** A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock *pxTaskWoken will get set to pdTRUE, otherwise *pxTaskWoken will remain unchanged.

Returns pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

BaseType_t **xQueueIsQueueEmptyFromISR** (const *QueueHandle_t* xQueue)

BaseType_t **xQueueIsQueueFullFromISR** (const *QueueHandle_t* xQueue)

UBaseType_t **uxQueueMessagesWaitingFromISR** (const *QueueHandle_t* xQueue)

void **vQueueAddToRegistry** (*QueueHandle_t* xQueue, const char *pcQueueName)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call vQueueAddToRegistry() add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger. If you are not using a kernel aware debugger then this function can be ignored.

configQUEUE_REGISTRY_SIZE defines the maximum number of handles the registry can hold. configQUEUE_REGISTRY_SIZE must be greater than 0 within FreeRTOSConfig.h for the registry to be available. Its value does not effect the number of queues, semaphores and mutexes that can be created - just the number that the registry can hold.

Parameters

- **xQueue** –The handle of the queue being added to the registry. This is the handle returned by a call to xQueueCreate(). Semaphore and mutex handles can also be passed in here.
- **pcQueueName** –The name to be associated with the handle. This is the name that the kernel aware debugger will display. The queue registry only stores a pointer to the string - so the string must be persistent (global or preferably in ROM/Flash), not on the stack.

void **vQueueUnregisterQueue**(*QueueHandle_t* xQueue)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call vQueueAddToRegistry() add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger, and vQueueUnregisterQueue() to remove the queue, semaphore or mutex from the register. If you are not using a kernel aware debugger then this function can be ignored.

Parameters **xQueue** –The handle of the queue being removed from the registry.

const char ***pcQueueGetName**(*QueueHandle_t* xQueue)

The queue registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call pcQueueGetName() to look up and return the name of a queue in the queue registry from the queue's handle.

Parameters **xQueue** –The handle of the queue the name of which will be returned.

Returns If the queue is in the registry then a pointer to the name of the queue is returned. If the queue is not in the registry then NULL is returned.

QueueHandle_t **xQueueGenericCreate**(const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize, const uint8_t ucQueueType)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

QueueHandle_t **xQueueGenericCreateStatic**(const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize, uint8_t *pucQueueStorage, StaticQueue_t *pxStaticQueue, const uint8_t ucQueueType)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

QueueSetHandle_t **xQueueCreateSet**(const UBaseType_t uxEventQueueLength)

Queue sets provide a mechanism to allow a task to block (pend) on a read operation from multiple queues or semaphores simultaneously.

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

A queue set must be explicitly created using a call to xQueueCreateSet() before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to xQueueAddToSet(). xQueueSelectFromSet() is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Note 1: See the documentation on https://www.FreeRTOS.org/RTOS-queue-sets.html for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: An additional 4 bytes of RAM is required for each space in a every queue added to a queue set. Therefore counting semaphores that have a high maximum count value should not be added to a queue set.

Note 4: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters **uxEventQueueLength** –Queue sets store events that occur on the queues and semaphores contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once. To be absolutely certain that events are not lost uxEventQueueLength should be set to the total sum of the length of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. Examples:

- If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18.
- If a queue set is to hold three binary semaphores then uxEventQueueLength should be set to (1 + 1 + 1), or 3.
- If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then uxEventQueueLength should be set to (5 + 3), or 8.

Returns If the queue set is created successfully then a handle to the created queue set is returned. Otherwise NULL is returned.

BaseType_t **xQueueAddToSet**(*QueueSetMemberHandle_t* xQueueOrSemaphore, *QueueSetHandle_t* xQueueSet)

Adds a queue or semaphore to a queue set that was previously created by a call to xQueueCreateSet().

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

Note 1: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters

- **xQueueOrSemaphore** –The handle of the queue or semaphore being added to the queue set (cast to an QueueSetMemberHandle_t type).
- **xQueueSet** –The handle of the queue set to which the queue or semaphore is being added.

Returns If the queue or semaphore was successfully added to the queue set then pdPASS is returned. If the queue could not be successfully added to the queue set because it is already a member of a different queue set then pdFAIL is returned.

BaseType_t **xQueueRemoveFromSet**(*QueueSetMemberHandle_t* xQueueOrSemaphore, *QueueSetHandle_t* xQueueSet)

Removes a queue or semaphore from a queue set. A queue or semaphore can only be removed from a set if the queue or semaphore is empty.

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

Parameters

- **xQueueOrSemaphore** –The handle of the queue or semaphore being removed from the queue set (cast to an QueueSetMemberHandle_t type).
- **xQueueSet** –The handle of the queue set in which the queue or semaphore is included.

Returns If the queue or semaphore was successfully removed from the queue set then pdPASS is returned. If the queue was not in the queue set, or the queue (or semaphore) was not empty, then pdFAIL is returned.

QueueSetMemberHandle_t **xQueueSelectFromSet**(*QueueSetHandle_t* xQueueSet, const TickType_t xTicksToWait)

xQueueSelectFromSet() selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). xQueueSelectFromSet() effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

Note 1: See the documentation on https://www.FreeRTOS.org/RTOS-queue-sets.html for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters

- **xQueueSet** –The queue set on which the task will (potentially) block.
- **xTicksToWait** –The maximum time, in ticks, that the calling task will remain in the Blocked state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

Returns xQueueSelectFromSet() will return the handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available, or NULL if no such queue or semaphore exists before before the specified block time expires.

QueueSetMemberHandle_t **xQueueSelectFromSetFromISR** (*QueueSetHandle_t* xQueueSet)

A version of xQueueSelectFromSet() that can be used from an ISR.

Macros

xQueueCreate (uxQueueLength, uxItemSize)

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the xQueueCreate() function. (see https://www.FreeRTOS.org/a00111.html). If a queue is created using xQueueCreateStatic() then the application writer must provide the memory that will get used by the queue. xQueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

https://www.FreeRTOS.org/Embedded-RTOS-Queues.html

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );
 if( xQueue1 == 0 )
 {
     // Queue was not created and must not be used.
 }

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
 if( xQueue2 == 0 )
 {
     // Queue was not created and must not be used.
 }

 // ... Rest of task code.
}
```

Parameters

- **uxQueueLength** –The maximum number of items that the queue can contain.
- **uxItemSize** –The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

xQueueCreateStatic(uxQueueLength, uxItemSize, pucQueueStorage, pxQueueBuffer)

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the xQueueCreate() function. (see https://www.FreeRTOS.org/a00111.html). If a queue is created using xQueueCreateStatic() then the application writer must provide the memory that will get used by the queue. xQueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

https://www.FreeRTOS.org/Embedded-RTOS-Queues.html

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
};

#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

// xQueueBuffer will hold the queue structure.
StaticQueue_t xQueueBuffer;

// ucQueueStorage will hold the items posted to the queue.  Must be at least
// [(queue length) * ( queue item size)] bytes long.
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can␣
↪hold.
                         ITEM_SIZE     // The size of each item in the queue
                         &( ucQueueStorage[ 0 ] ), // The buffer that will␣
↪hold the items in the queue.
                         &xQueueBuffer ); // The buffer that will hold the␣
↪queue structure.

 // The queue is guaranteed to be created successfully as no dynamic memory
 // allocation is used.  Therefore xQueue1 is now a handle to a valid queue.

 // ... Rest of task code.
}
```

Parameters

- **uxQueueLength** –The maximum number of items that the queue can contain.
- **uxItemSize** –The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
- **pucQueueStorage** –If uxItemSize is not zero then pucQueueStorageBuffer must point to a uint8_t array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is (uxQueueLength * uxItemsSize) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL.

- **pxQueueBuffer** –Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

Returns If the queue is created then a handle to the created queue is returned. If pxQueueBuffer is NULL then NULL is returned.

xQueueSendToFront(xQueue, pvItemToQueue, xTicksToWait)

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

 // ...

 if( xQueue1 != 0 )
 {
     // Send an uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSendToFront( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) !=␣
↪pdPASS )
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

 if( xQueue2 != 0 )
 {
     // Send a pointer to a struct AMessage object.  Don't block if the
     // queue is already full.
     pxMessage = & xMessage;
     xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
 }

 // ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes

will be copied from pvItemToQueue into the queue storage area.

- **xTicksToWait** –The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

xQueueSendToBack (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls xQueueGenericSend().

Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

 // ...

 if( xQueue1 != 0 )
 {
     // Send an uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) !=␣
↪pdPASS )
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

 if( xQueue2 != 0 )
 {
     // Send a pointer to a struct AMessage object.  Don't block if the
     // queue is already full.
     pxMessage = & xMessage;
     xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
 }

 // ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **xTicksToWait** –The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

xQueueSend(xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls xQueueGenericSend(). It is included for backward compatibility with versions of FreeRTOS.org that did not include the xQueueSendToFront() and xQueueSendToBack() macros. It is equivalent to xQueueSendToBack().

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
 char ucMessageID;
 char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // Create a queue capable of containing 10 pointers to AMessage structures.
 // These should be passed by pointer as they contain a lot of data.
 xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

 // ...

 if( xQueue1 != 0 )
 {
     // Send an uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS␣
↪)
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

 if( xQueue2 != 0 )
 {
     // Send a pointer to a struct AMessage object.  Don't block if the
     // queue is already full.
     pxMessage = & xMessage;
     xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
 }
```

(continues on next page)

(continued from previous page)

```
 // ... Rest of task code.
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **xTicksToWait** –The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

Returns pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

xQueueOverwrite(xQueue, pvItemToQueue)

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

Example usage:

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
uint32_t ulVarToSend, ulValReceived;

 // Create a queue to hold one uint32_t value.  It is strongly
 // recommended *not* to use xQueueOverwrite() on queues that can
 // contain more than one value, and doing so will trigger an assertion
 // if configASSERT() is defined.
 xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

 // Write the value 10 to the queue using xQueueOverwrite().
 ulVarToSend = 10;
 xQueueOverwrite( xQueue, &ulVarToSend );

 // Peeking the queue should now return 10, but leave the value 10 in
 // the queue.  A block time of zero is used as it is known that the
 // queue holds a value.
 ulValReceived = 0;
 xQueuePeek( xQueue, &ulValReceived, 0 );

 if( ulValReceived != 10 )
 {
     // Error unless the item was removed by a different task.
 }

 // The queue is still full.  Use xQueueOverwrite() to overwrite the
 // value held in the queue with 100.
 ulVarToSend = 100;
 xQueueOverwrite( xQueue, &ulVarToSend );

 // This time read from the queue, leaving the queue empty once more.
```

(continues on next page)

(continued from previous page)

```
 // A block time of 0 is used again.
 xQueueReceive( xQueue, &ulValReceived, 0 );

 // The value read should be the last value written, even though the
 // queue was already full when the value was written.
 if( ulValReceived != 100 )
 {
     // Error!
 }

 // ...
}
```

Parameters

- **xQueue** –The handle of the queue to which the data is being sent.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

Returns xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

xQueueSendToFrontFromISR (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPrioritTaskWoken;

 // We have not woken a task at the start of the ISR.
 xHigherPriorityTaskWoken = pdFALSE;

 // Loop until the buffer is empty.
 do
 {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

     // Post the byte.
     xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

 } while( portINPUT_BYTE( BUFFER_COUNT ) );

 // Now the buffer is empty we can switch context if necessary.
 if( xHigherPriorityTaskWoken )
 {
     portYIELD_FROM_ISR ();
 }
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **pxHigherPriorityTaskWoken** –**[out]** xQueueSendToFrontFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToFromFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

xQueueSendToBackFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the back of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

 // We have not woken a task at the start of the ISR.
 xHigherPriorityTaskWoken = pdFALSE;

 // Loop until the buffer is empty.
 do
 {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

     // Post the byte.
     xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

 } while( portINPUT_BYTE( BUFFER_COUNT ) );

 // Now the buffer is empty we can switch context if necessary.
 if( xHigherPriorityTaskWoken )
 {
     portYIELD_FROM_ISR ();
 }
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **pxHigherPriorityTaskWoken** –**[out]** xQueueSendToBackFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToBackFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

xQueueOverwriteFromISR (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

A version of xQueueOverwrite() that can be used in an interrupt service routine (ISR).

Only for use with queues that can hold a single item - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

Example usage:

```
QueueHandle_t xQueue;

void vFunction( void *pvParameters )
{
 // Create a queue to hold one uint32_t value.  It is strongly
 // recommended *not* to use xQueueOverwriteFromISR() on queues that can
 // contain more than one value, and doing so will trigger an assertion
 // if configASSERT() is defined.
 xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}

void vAnInterruptHandler( void )
{
// xHigherPriorityTaskWoken must be set to pdFALSE before it is used.
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
uint32_t ulVarToSend, ulValReceived;

 // Write the value 10 to the queue using xQueueOverwriteFromISR().
 ulVarToSend = 10;
 xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

 // The queue is full, but calling xQueueOverwriteFromISR() again will still
 // pass because the value held in the queue will be overwritten with the
 // new value.
 ulVarToSend = 100;
 xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

 // Reading from the queue will now return 100.

 // ...

 if( xHigherPrioritytaskWoken == pdTRUE )
 {
     // Writing to the queue caused a task to unblock and the unblocked task
     // has a priority higher than or equal to the priority of the currently
     // executing task (the task this interrupt interrupted).  Perform a␣
↪context
     // switch so this interrupt returns directly to the unblocked task.
     portYIELD_FROM_ISR(); // or portEND_SWITCHING_ISR() depending on the port.
 }
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **pxHigherPriorityTaskWoken** –**[out]** xQueueOverwriteFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be

requested before the interrupt is exited.

Returns xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

xQueueSendFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR(). It is included for backward compatibility with versions of FreeRTOS.org that did not include the xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() macros.

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

 // We have not woken a task at the start of the ISR.
 xHigherPriorityTaskWoken = pdFALSE;

 // Loop until the buffer is empty.
 do
 {
     // Obtain a byte from the buffer.
     cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

     // Post the byte.
     xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

 } while( portINPUT_BYTE( BUFFER_COUNT ) );

 // Now the buffer is empty we can switch context if necessary.
 if( xHigherPriorityTaskWoken )
 {
     // Actual macro used here is port specific.
     portYIELD_FROM_ISR ();
 }
}
```

Parameters

- **xQueue** –The handle to the queue on which the item is to be posted.
- **pvItemToQueue** –A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **pxHigherPriorityTaskWoken** –**[out]** xQueueSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

xQueueReset(xQueue)

Reset a queue back to its original empty state. The return value is now obsolete and is always set to pdPASS.

Type Definitions

typedef struct QueueDefinition ***QueueHandle_t**

typedef struct QueueDefinition ***QueueSetHandle_t**

Type by which queue sets are referenced. For example, a call to xQueueCreateSet() returns an xQueueSet variable that can then be used as a parameter to xQueueSelectFromSet(), xQueueAddToSet(), etc.

typedef struct QueueDefinition ***QueueSetMemberHandle_t**

Queue sets can contain both queues and semaphores, so the QueueSetMemberHandle_t is defined as a type to be used where a parameter or return value can be either an QueueHandle_t or an SemaphoreHandle_t.

Semaphore API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/semphr.h

Macros

semBINARY_SEMAPHORE_QUEUE_LENGTH

semSEMAPHORE_QUEUE_ITEM_LENGTH

semGIVE_BLOCK_TIME

vSemaphoreCreateBinary(xSemaphore)

xSemaphoreCreateBinary()

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! https://www.FreeRTOS.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see https://www.FreeRTOS.org/a00111.html). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

The old vSemaphoreCreateBinary() macro is now deprecated in favour of this xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreateBinary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
 // Semaphore cannot be used before a call to vSemaphoreCreateBinary().
 // This is a macro so pass the variable in directly.
 xSemaphore = xSemaphoreCreateBinary();

 if( xSemaphore != NULL )
 {
     // The semaphore was created successfully.
     // The semaphore can now be used.
 }
}
```

Returns Handle to the created semaphore, or NULL if the memory required to hold the semaphore's data structures could not be allocated.

xSemaphoreCreateBinaryStatic(pxStaticSemaphore)

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

NOTE: In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! https://www.FreeRTOS.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see https://www.FreeRTOS.org/a00111.html). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
 // Semaphore cannot be used before a call to xSemaphoreCreateBinary() or
 // xSemaphoreCreateBinaryStatic().
 // The semaphore's data structures will be placed in the xSemaphoreBuffer
 // variable, the address of which is passed into the function.  The
 // function's parameter is not NULL, so the function will not attempt any
 // dynamic memory allocation, and therefore the function will not return
 // return NULL.
 xSemaphore = xSemaphoreCreateBinaryStatic( &xSemaphoreBuffer );

 // Rest of task code goes here.
}
```

Parameters

- **pxStaticSemaphore** –Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

Returns If the semaphore is created then a handle to the created semaphore is returned. If pxSemaphoreBuffer is NULL then NULL is returned.

xSemaphoreTake (xSemaphore, xBlockTime)

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

param xSemaphore A handle to the semaphore being taken - obtained when the semaphore was created.

param xBlockTime The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
 // Create the semaphore to guard a shared resource.
 vSemaphoreCreateBinary( xSemaphore );
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
 // ... Do other things.

 if( xSemaphore != NULL )
 {
     // See if we can obtain the semaphore.  If the semaphore is not available
     // wait 10 ticks to see if it becomes free.
     if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
     {
         // We were able to obtain the semaphore and can now access the
         // shared resource.

         // ...

         // We have finished accessing the shared resource.  Release the
         // semaphore.
         xSemaphoreGive( xSemaphore );
     }
     else
     {
         // We could not obtain the semaphore and can therefore not access
         // the shared resource safely.
     }
 }
}
```

Returns pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

xSemaphoreTakeRecursive (xMutex, xBlockTime)

Macro to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
 // Create the mutex to guard a shared resource.
 xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
 // ... Do other things.

 if( xMutex != NULL )
 {
     // See if we can obtain the mutex.  If the mutex is not available
     // wait 10 ticks to see if it becomes free.
     if( xSemaphoreTakeRecursive( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
     {
         // We were able to obtain the mutex and can now access the
         // shared resource.

         // ...
         // For some reason due to the nature of the code further calls to
         // xSemaphoreTakeRecursive() are made on the same mutex.  In real
         // code these would not be just sequential calls as this would make
         // no sense.  Instead the calls are likely to be buried inside
         // a more complex call structure.
         xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
         xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

         // The mutex has now been 'taken' three times, so will not be
         // available to another task until it has also been given back
         // three times.  Again it is unlikely that real code would have
         // these calls sequentially, but instead buried in a more complex
         // call structure.  This is just for illustrative purposes.
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );

         // Now the mutex can be taken by other tasks.
     }
     else
     {
         // We could not obtain the mutex and can therefore not access
         // the shared resource safely.
     }
 }
}
```

Parameters

- **xMutex** –A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex();
- **xBlockTime** –The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xBlock-Time.

Returns pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

xSemaphoreGive(xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphoreTake().

This macro must not be used from an ISR. See xSemaphoreGiveFromISR () for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
 // Create the semaphore to guard a shared resource.
 vSemaphoreCreateBinary( xSemaphore );

 if( xSemaphore != NULL )
 {
     if( xSemaphoreGive( xSemaphore ) != pdTRUE )
     {
         // We would expect this call to fail because we cannot give
         // a semaphore without first "taking" it!
     }

     // Obtain the semaphore - don't block if the semaphore is not
     // immediately available.
     if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
     {
         // We now have the semaphore and can access the shared resource.

         // ...

         // We have finished accessing the shared resource so can free the
         // semaphore.
         if( xSemaphoreGive( xSemaphore ) != pdTRUE )
         {
             // We would not expect this call to fail because we must have
             // obtained the semaphore to get here.
         }
     }
 }
}
```

Parameters

- **xSemaphore** –A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

xSemaphoreGiveRecursive (xMutex)

Macro to recursively release, or ‘give’, a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called xSemaphoreGiveRecursive() for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
 // Create the mutex to guard a shared resource.
 xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
 // ... Do other things.

 if( xMutex != NULL )
 {
     // See if we can obtain the mutex.  If the mutex is not available
     // wait 10 ticks to see if it becomes free.
     if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
     {
         // We were able to obtain the mutex and can now access the
         // shared resource.

         // ...
         // For some reason due to the nature of the code further calls to
         // xSemaphoreTakeRecursive() are made on the same mutex.  In real
         // code these would not be just sequential calls as this would make
         // no sense.  Instead the calls are likely to be buried inside
         // a more complex call structure.
         xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
         xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

         // The mutex has now been 'taken' three times, so will not be
         // available to another task until it has also been given back
         // three times.  Again it is unlikely that real code would have
         // these calls sequentially, it would be more likely that the calls
         // to xSemaphoreGiveRecursive() would be called as a call stack
         // unwound.  This is just for demonstrative purposes.
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );
         xSemaphoreGiveRecursive( xMutex );
```

(continues on next page)

(continued from previous page)

```
            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

Parameters

- **xMutex** –A handle to the mutex being released, or ‘given’. This is the handle returned by xSemaphoreCreateMutex();

Returns pdTRUE if the semaphore was given.

xSemaphoreGiveFromISR (xSemaphore, pxHigherPriorityTaskWoken)

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.

Example usage:

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT  10
SemaphoreHandle_t xSemaphore = NULL;

// Repetitive task.
void vATask( void * pvParameters )
{
    for( ;; )
    {
        // We want this task to run every 10 ticks of a timer.  The semaphore
        // was created before this task was started.

        // Block waiting for the semaphore to become available.
        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
        {
            // It is time to execute.

            // ...

            // We have finished our task.  Return to the top of the loop where
            // we will block on the semaphore until it is time to execute
            // again.  Note when using the semaphore for synchronisation with an
            // ISR in this manner there is no need to 'give' the semaphore back.
        }
    }
}

// Timer ISR
void vTimerISR( void * pvParameters )
{
static uint8_t ucLocalTickCount = 0;
static BaseType_t xHigherPriorityTaskWoken;
```

(continues on next page)

(continued from previous page)

```
// A timer tick has occurred.

// ... Do other time functions.

// Is it time for vATask () to run?
xHigherPriorityTaskWoken = pdFALSE;
ucLocalTickCount++;
if( ucLocalTickCount >= TICKS_TO_WAIT )
{
    // Unblock the task by releasing the semaphore.
    xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

    // Reset the count so we release the semaphore again in 10 ticks time.
    ucLocalTickCount = 0;
}

if( xHigherPriorityTaskWoken != pdFALSE )
{
    // We can force a context switch here.  Context switching from an
    // ISR uses port specific syntax.  Check the demo task for your port
    // to find the syntax required.
}
}
```

Parameters

- **xSemaphore** –A handle to the semaphore being released. This is the handle returned when the semaphore was created.
- **pxHigherPriorityTaskWoken** –xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

xSemaphoreTakeFromISR(xSemaphore, pxHigherPriorityTaskWoken)

Macro to take a semaphore from an ISR. The semaphore must have previously been created with a call to xSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR, however taking a semaphore from an ISR is not a common operation. It is likely to only be useful when taking a counting semaphore when an interrupt is obtaining an object from a resource pool (when the semaphore count indicates the number of resources available).

Parameters

- **xSemaphore** –A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
- **pxHigherPriorityTaskWoken** –**[out]** xSemaphoreTakeFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if taking the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreTakeFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

xSemaphoreCreateMutex()

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required

memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see https://www.FreeRTOS.org/a00111.html). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
 // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
 // This is a macro so pass the variable in directly.
 xSemaphore = xSemaphoreCreateMutex();

 if( xSemaphore != NULL )
 {
     // The semaphore was created successfully.
     // The semaphore can now be used.
 }
}
```

Returns If the mutex was successfully created then a handle to the created semaphore is returned. If there was not enough heap to allocate the mutex data structures then NULL is returned.

xSemaphoreCreateMutexStatic(pxMutexBuffer)

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see https://www.FreeRTOS.org/a00111.html). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
 // A mutex cannot be used before it has been created.  xMutexBuffer is
 // into xSemaphoreCreateMutexStatic() so no dynamic memory allocation is
 // attempted.
 xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );

 // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
 // so there is no need to check it.
}
```

Parameters

- **pxMutexBuffer** –Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's data structure, removing the need for the memory to be allocated dynamically.

Returns If the mutex was successfully created then a handle to the created mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

xSemaphoreCreateCounting(uxMaxCount, uxInitialCount)

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see http://www.freertos.org/a00111.html). If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. xSemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros. The xSemaphoreTake() and xSemaphoreGive() macros must not be used.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See vSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
   // This is a macro so pass the variable in directly.
   xSemaphore = xSemaphoreCreateRecursiveMutex();
```

(continues on next page)

(continued from previous page)

```
    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see https://www.FreeRTOS.org/a00111.html). If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. xSemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros. The xSemaphoreTake() and xSemaphoreGive() macros must not be used.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
   // A recursive semaphore cannot be used before it is created.  Here a
   // recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic().
   // The address of xMutexBuffer is passed into the function, and will hold
   // the mutexes data structures - so no dynamic memory allocation will be
   // attempted.
   xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );

   // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
   // so there is no need to check it.
}
```

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! https://www.FreeRTOS.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreate-

Counting() function. (see https://www.FreeRTOS.org/a00111.html). If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the application writer can instead optionally provide the memory that will get used by the counting semaphore. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore = NULL;

 // Semaphore cannot be used before a call to xSemaphoreCreateCounting().
 // The max value to which the semaphore can count should be 10, and the
 // initial value assigned to the count should be 0.
 xSemaphore = xSemaphoreCreateCounting( 10, 0 );

 if( xSemaphore != NULL )
 {
     // The semaphore was created successfully.
     // The semaphore can now be used.
 }
}
```

Returns xSemaphore Handle to the created mutex semaphore. Should be of type SemaphoreHandle_t.

Parameters

- **pxStaticSemaphore** –Must point to a variable of type StaticSemaphore_t, which will then be used to hold the recursive mutex's data structure, removing the need for the memory to be allocated dynamically.
- **uxMaxCount** –The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.
- **uxInitialCount** –The count value assigned to the semaphore when it is created.

Returns If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Returns Handle to the created semaphore. Null if the semaphore could not be created.

xSemaphoreCreateCountingStatic(uxMaxCount, uxInitialCount, pxSemaphoreBuffer)

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! https://www.FreeRTOS.org/RTOS-task-notifications.html

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see https://www.FreeRTOS.org/a00111.html). If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the application writer must provide the memory. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore = NULL;

 // Counting semaphore cannot be used before they have been created.  Create
 // a counting semaphore using xSemaphoreCreateCountingStatic().  The max
 // value to which the semaphore can count is 10, and the initial value
 // assigned to the count will be 0.  The address of xSemaphoreBuffer is
 // passed in and will be used to hold the semaphore structure, so no dynamic
 // memory allocation will be used.
 xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );

 // No memory allocation was attempted so xSemaphore cannot be NULL, so there
 // is no need to check its value.
}
```

Parameters

- **uxMaxCount** –The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.
- **uxInitialCount** –The count value assigned to the semaphore when it is created.
- **pxSemaphoreBuffer** –Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

Returns If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If pxSemaphoreBuffer was NULL then NULL is returned.

vSemaphoreDelete (xSemaphore)

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore if the mutex is held by a task.

Parameters

- **xSemaphore** –A handle to the semaphore to be deleted.

xSemaphoreGetMutexHolder (xSemaphore)

If xMutex is indeed a mutex type semaphore, return the current mutex holder. If xMutex is not a mutex type semaphore, or the mutex is available (not held by a task), return NULL.

Note: This is a good way of determining if the calling task is the mutex holder, but not a good way of determining the identity of the mutex holder as the holder may change between the function exiting and the returned value being tested.

xSemaphoreGetMutexHolderFromISR (xSemaphore)

If xMutex is indeed a mutex type semaphore, return the current mutex holder. If xMutex is not a mutex type semaphore, or the mutex is available (not held by a task), return NULL.

uxSemaphoreGetCount (xSemaphore)

If the semaphore is a counting semaphore then uxSemaphoreGetCount() returns its current count value. If the semaphore is a binary semaphore then uxSemaphoreGetCount() returns 1 if the semaphore is available, and 0 if the semaphore is not available.

Type Definitions

typedef *QueueHandle_t* **SemaphoreHandle_t**

Timer API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/timers.h

Functions

TimerHandle_t **xTimerCreate** (const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *const pvTimerID, *TimerCallbackFunction_t* pxCallbackFunction)

TimerHandle_t xTimerCreate(const char * const pcTimerName, TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction);

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using xTimerCreate() then the required memory is automatically dynamically allocated inside the xTimerCreate() function. (see https://www.FreeRTOS.org/a00111.html). If a software timer is created using xTimerCreateStatic() then the application writer must provide the memory that will get used by the software timer. xTimerCreateStatic() therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
* #define NUM_TIMERS 5
*
* // An array to hold handles to the created timers.
* TimerHandle_t xTimers[ NUM_TIMERS ];
*
* // An array to hold a count of the number of times each timer expires.
* int32_t lExpireCounters[ NUM_TIMERS ] = { 0 };
*
* // Define a callback function that will be used by multiple timer instances.
* // The callback function does nothing but count the number of times the
* // associated timer expires, and stop the timer once the timer has expired
* // 10 times.
* void vTimerCallback( TimerHandle_t pxTimer )
* {
* int32_t lArrayIndex;
* const int32_t xMaxExpiryCountBeforeStopping = 10;
*
*     // Optionally do something if the pxTimer parameter is NULL.
*     configASSERT( pxTimer );
*
*     // Which timer expired?
*     lArrayIndex = ( int32_t ) pvTimerGetTimerID( pxTimer );
*
*     // Increment the number of times that pxTimer has expired.
*     lExpireCounters[ lArrayIndex ] += 1;
*
*     // If the timer has expired 10 times then stop it from running.
*     if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
*     {
*         // Do not use a block time if calling a timer API function from a
*         // timer callback function, as doing so could cause a deadlock!
*         xTimerStop( pxTimer, 0 );
*     }
* }
*
* void main( void )
* {
* int32_t x;
*
*     // Create then start some timers.  Starting the timers before the␣
↪scheduler
*     // has been started means the timers will start running immediately that
*     // the scheduler starts.
*     for( x = 0; x < NUM_TIMERS; x++ )
*     {
*         xTimers[ x ] = xTimerCreate(    "Timer",       // Just a text name,␣
↪not used by the kernel.
*                                         ( 100 * x ),   // The timer period␣
↪in ticks.
*                                         pdTRUE,        // The timers will␣
↪auto-reload themselves when they expire.
*                                         ( void * ) x,  // Assign each timer␣
↪a unique id equal to its array index.
*                                         vTimerCallback // Each timer calls␣
↪the same callback when it expires.
*                                       );
*
*         if( xTimers[ x ] == NULL )
*         {
*             // The timer was not created.
*         }
*         else
```

(continues on next page)

(continued from previous page)

```
*          {
*              // Start the timer.  No block time is specified, and even if one␣
↪was
*              // it would be ignored because the scheduler has not yet been
*              // started.
*              if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
*              {
*                  // The timer could not be set into the Active state.
*              }
*          }
*      }
*
*      // ...
*      // Create tasks here.
*      // ...
*
*      // Starting the scheduler will start the timers running as they have␣
↪already
*      // been set into the active state.
*      vTaskStartScheduler();
*
*      // Should not reach here.
*      for( ;; );
* }
*
```

Parameters

- **pcTimerName** –A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- **xTimerPeriodInTicks** –The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000. Time timer period must be greater than 0.
- **uxAutoReload** –If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
- **pvTimerID** –An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- **pxCallbackFunction** –The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);" .

Returns If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then NULL is returned.

TimerHandle_t **xTimerCreateStatic**(const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *const pvTimerID, *TimerCallbackFunction_t* pxCallbackFunction, StaticTimer_t *pxTimerBuffer)

TimerHandle_t xTimerCreateStatic(const char * const pcTimerName, TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction, StaticTimer_t *pxTimerBuffer);

Creates a new software timer instance, and returns a handle by which the created software timer can be refer-

enced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using xTimerCreate() then the required memory is automatically dynamically allocated inside the xTimerCreate() function. (see https://www.FreeRTOS.org/a00111.html). If a software timer is created using xTimerCreateStatic() then the application writer must provide the memory that will get used by the software timer. xTimerCreateStatic() therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
*
* // The buffer used to hold the software timer's data structure.
* static StaticTimer_t xTimerBuffer;
*
* // A variable that will be incremented by the software timer's callback
* // function.
* UBaseType_t uxVariableToIncrement = 0;
*
* // A software timer callback function that increments a variable passed to
* // it when the software timer was created.  After the 5th increment the
* // callback function stops the software timer.
* static void prvTimerCallback( TimerHandle_t xExpiredTimer )
* {
* UBaseType_t *puxVariableToIncrement;
* BaseType_t xReturned;
*
*     // Obtain the address of the variable to increment from the timer ID.
*     puxVariableToIncrement = ( UBaseType_t * ) pvTimerGetTimerID(␣
↪xExpiredTimer );
*
*     // Increment the variable to show the timer callback has executed.
*     ( *puxVariableToIncrement )++;
*
*     // If this callback has executed the required number of times, stop the
*     // timer.
*     if( *puxVariableToIncrement == 5 )
*     {
*         // This is called from a timer callback so must not block.
*         xTimerStop( xExpiredTimer, staticDONT_BLOCK );
*     }
* }
*
*
* void main( void )
* {
*     // Create the software time.  xTimerCreateStatic() has an extra parameter
*     // than the normal xTimerCreate() API function.  The parameter is a␣
↪pointer
*     // to the StaticTimer_t structure that will hold the software timer
*     // structure.  If the parameter is passed as NULL then the structure␣
↪will be
*     // allocated dynamically, just as if xTimerCreate() had been called.
*     xTimer = xTimerCreateStatic( "T1",             // Text name for the task.
↪  Helps debugging only.  Not used by FreeRTOS.
*                                  xTimerPeriod,     // The period of the␣
↪timer in ticks.
```

(continues on next page)

(continued from previous page)

```
*                                   pdTRUE,           // This is an auto-reload␣
↪timer.
*                                   ( void * ) &uxVariableToIncrement,    // A␣
↪variable incremented by the software timer's callback function
*                                   prvTimerCallback, // The function to␣
↪execute when the timer expires.
*                                   &xTimerBuffer );  // The buffer that will␣
↪hold the software timer structure.
*
*     // The scheduler has not started yet so a block time is not used.
*     xReturned = xTimerStart( xTimer, 0 );
*
*     // ...
*     // Create tasks here.
*     // ...
*
*     // Starting the scheduler will start the timers running as they have␣
↪already
*     // been set into the active state.
*     vTaskStartScheduler();
*
*     // Should not reach here.
*     for( ;; );
* }
*
```

Parameters

- **pcTimerName** –A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- **xTimerPeriodInTicks** –The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000. The timer period must be greater than 0.
- **uxAutoReload** –If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
- **pvTimerID** –An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- **pxCallbackFunction** –The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);" .
- **pxTimerBuffer** –Must point to a variable of type StaticTimer_t, which will be then be used to hold the software timer's data structures, removing the need for the memory to be allocated dynamically.

Returns If the timer is created then a handle to the created timer is returned. If pxTimerBuffer was NULL then NULL is returned.

void ***pvTimerGetTimerID**(const *TimerHandle_t* xTimer)

void *pvTimerGetTimerID(TimerHandle_t xTimer);

Returns the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer, and by calling the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Example usage:

See the xTimerCreate() API function example usage scenario.

Parameters **xTimer** –The timer being queried.
Returns The ID assigned to the timer being queried.

void **vTimerSetTimerID**(*TimerHandle_t* xTimer, void *pvNewID)

void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID);

Sets the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Example usage:

See the xTimerCreate() API function example usage scenario.

Parameters
- **xTimer** –The timer being updated.
- **pvNewID** –The ID to assign to the timer.

BaseType_t **xTimerIsTimerActive**(*TimerHandle_t* xTimer)

BaseType_t xTimerIsTimerActive(TimerHandle_t xTimer);

Queries a timer to see if it is active or dormant.

A timer will be dormant if: 1) It has been created but not started, or 2) It is an expired one-shot timer that has not been restarted.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
* // This function assumes xTimer has already been created.
* void vAFunction( TimerHandle_t xTimer )
* {
*     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and␣
↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
*     {
*         // xTimer is active, do something.
*     }
*     else
*     {
*         // xTimer is not active, do something else.
*     }
* }
*
```

Parameters **xTimer** –The timer being queried.
Returns pdFALSE will be returned if the timer is dormant. A value other than pdFALSE will be returned if the timer is active.

TaskHandle_t **xTimerGetTimerDaemonTaskHandle** (void)

TaskHandle_t xTimerGetTimerDaemonTaskHandle(void);

Simply returns the handle of the timer service/daemon task. It it not valid to call xTimerGetTimerDaemonTaskHandle() before the scheduler has been started.

BaseType_t **xTimerPendFunctionCallFromISR** (*PendedFunction_t* xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken)

BaseType_t xTimerPendFunctionCallFromISR(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken);

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended callback function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.

Example usage:

```
*
*  // The callback function that will execute in the context of the daemon␣
↪task.
*  // Note callback functions must all use this same prototype.
*  void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
*  {
*      BaseType_t xInterfaceToService;
*
*      // The interface that requires servicing is passed in the second
*      // parameter.  The first parameter is not used in this case.
*      xInterfaceToService = ( BaseType_t ) ulParameter2;
*
*      // ...Perform the processing here...
*  }
*
*  // An ISR that receives data packets from multiple interfaces
*  void vAnISR( void )
*  {
*      BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;
*
*      // Query the hardware to determine which interface needs processing.
*      xInterfaceToService = prvCheckInterfaces();
*
*      // The actual processing is to be deferred to a task.  Request the
*      // vProcessInterface() callback function is executed, passing in the
*      // number of the interface that needs processing.  The interface to
*      // service is passed in the second parameter.  The first parameter is
*      // not used in this case.
*      xHigherPriorityTaskWoken = pdFALSE;
*      xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t )␣
↪xInterfaceToService, &xHigherPriorityTaskWoken );
*
*      // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
*      // switch should be requested.  The macro used is port specific and will
*      // be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
*      // the documentation page for the port being used.
*      portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

(continues on next page)

(continued from previous page)

```
*
*   }
*
```

Parameters

- **xFunctionToPend** –The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
- **pvParameter1** –The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- **ulParameter2** –The value of the callback function's second parameter.
- **pxHigherPriorityTaskWoken** –As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task (which is set using configTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

BaseType_t **xTimerPendFunctionCall**(*PendedFunction_t* xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait)

BaseType_t xTimerPendFunctionCall(PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait);

Used to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Parameters

- **xFunctionToPend** –The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
- **pvParameter1** –The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- **ulParameter2** –The value of the callback function's second parameter.
- **xTicksToWait** –Calling this function will result in a message being sent to the timer daemon task on a queue. xTicksToWait is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full.

Returns pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

const char ***pcTimerGetName**(*TimerHandle_t* xTimer)

const char * const pcTimerGetName(TimerHandle_t xTimer);

Returns the name that was assigned to a timer when the timer was created.

Parameters **xTimer** –The handle of the timer being queried.

Returns The name assigned to the timer specified by the xTimer parameter.

void **vTimerSetReloadMode**(*TimerHandle_t* xTimer, const UBaseType_t uxAutoReload)

void vTimerSetReloadMode(TimerHandle_t xTimer, const UBaseType_t uxAutoReload);

Updates a timer to be either an auto-reload timer, in which case the timer automatically resets itself each time it expires, or a one-shot timer, in which case the timer will only expire once unless it is manually restarted.

Parameters

- **xTimer** –The handle of the timer being updated.

- **uxAutoReload** –If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the timer's period (see the xTimerPeriodInTicks parameter of the xTimerCreate() API function). If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.

UBaseType_t **uxTimerGetReloadMode**(*TimerHandle_t* xTimer)

UBaseType_t uxTimerGetReloadMode(TimerHandle_t xTimer);

Queries a timer to determine if it is an auto-reload timer, in which case the timer automatically resets itself each time it expires, or a one-shot timer, in which case the timer will only expire once unless it is manually restarted.

Parameters **xTimer** –The handle of the timer being queried.
Returns If the timer is an auto-reload timer then pdTRUE is returned, otherwise pdFALSE is returned.

TickType_t **xTimerGetPeriod**(*TimerHandle_t* xTimer)

TickType_t xTimerGetPeriod(TimerHandle_t xTimer);

Returns the period of a timer.

Parameters **xTimer** –The handle of the timer being queried.
Returns The period of the timer in ticks.

TickType_t **xTimerGetExpiryTime**(*TimerHandle_t* xTimer)

TickType_t xTimerGetExpiryTime(TimerHandle_t xTimer);

Returns the time in ticks at which the timer will expire. If this is less than the current tick count then the expiry time has overflowed from the current time.

Parameters **xTimer** –The handle of the timer being queried.
Returns If the timer is running then the time in ticks at which the timer will next expire is returned. If the timer is not running then the return value is undefined.

void **vApplicationGetTimerTaskMemory**(StaticTask_t **ppxTimerTaskTCBBuffer, StackType_t **ppxTimerTaskStackBuffer, uint32_t *pulTimerTaskStackSize)

This function is used to provide a statically allocated block of memory to FreeRTOS to hold the Timer Task TCB. This function is required when configSUPPORT_STATIC_ALLOCATION is set. For more information see this URI: https://www.FreeRTOS.org/a00110.html#configSUPPORT_STATIC_ALLOCATION

Parameters

- **ppxTimerTaskTCBBuffer** –A handle to a statically allocated TCB buffer
- **ppxTimerTaskStackBuffer** –A handle to a statically allocated Stack buffer for thie idle task
- **pulTimerTaskStackSize** –A pointer to the number of elements that will fit in the allocated stack buffer

Macros

tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR

tmrCOMMAND_EXECUTE_CALLBACK

tmrCOMMAND_START_DONT_TRACE

tmrCOMMAND_START

tmrCOMMAND_RESET

tmrCOMMAND_STOP

tmrCOMMAND_CHANGE_PERIOD

tmrCOMMAND_DELETE

tmrFIRST_FROM_ISR_COMMAND

tmrCOMMAND_START_FROM_ISR

tmrCOMMAND_RESET_FROM_ISR

tmrCOMMAND_STOP_FROM_ISR

tmrCOMMAND_CHANGE_PERIOD_FROM_ISR

xTimerStart (xTimer, xTicksToWait)

BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerStart() was called, where 'n' is the timers defined period.

It is valid to call xTimerStart() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerStart() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

Example usage:

See the xTimerCreate() API function example usage scenario.

Parameters

- **xTimer** –The handle of the timer being started/restarted.
- **xTicksToWait** –Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

Returns pdFAIL will be returned if the start command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerStop (xTimer, xTicksToWait)

BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStop() stops a timer that was previously started using either of the The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() or xTimerChangePeriodFromISR() API functions.

Stopping a timer ensures the timer is not in the active state.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStop() to be available.

Example usage:

See the xTimerCreate() API function example usage scenario.

Parameters

- **xTimer** –The handle of the timer being stopped.
- **xTicksToWait** –Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when xTimerStop() was called. xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

Returns pdFAIL will be returned if the stop command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerChangePeriod (xTimer, xNewPeriod, xTicksToWait)

BaseType_t xTimerChangePeriod(TimerHandle_t xTimer, TickType_t xNewPeriod, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerChangePeriod() changes the period of a timer that was previously created using the xTimerCreate() API function.

xTimerChangePeriod() can be called to change the period of an active or dormant state timer.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Example usage:

```
* // This function assumes xTimer has already been created.  If the timer
* // referenced by xTimer is already active when it is called, then the timer
* // is deleted.  If the timer referenced by xTimer is not active when it is
* // called, then the period of the timer is set to 500ms and the timer is
* // started.
* void vAFunction( TimerHandle_t xTimer )
* {
*     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and␣
↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
*     {
```

(continues on next page)

(continued from previous page)

```
*          // xTimer is already active - delete it.
*          xTimerDelete( xTimer );
*      }
*      else
*      {
*          // xTimer is not active, change its period to 500ms.  This will also
*          // cause the timer to start.  Block for a maximum of 100 ticks if the
*          // change period command cannot immediately be sent to the timer
*          // command queue.
*          if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 ) ==␣
↪pdPASS )
*          {
*              // The command was successfully sent.
*          }
*          else
*          {
*              // The command could not be sent, even after waiting for 100␣
↪ticks
*              // to pass.  Take appropriate action here.
*          }
*      }
* }
*
```

Parameters

- **xTimer** –The handle of the timer that is having its period changed.
- **xNewPeriod** –The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- **xTicksToWait** –Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

Returns pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerDelete(xTimer, xTicksToWait)

BaseType_t xTimerDelete(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerDelete() deletes a timer that was previously created using the xTimerCreate() API function.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerDelete() to be available.

Example usage:

See the xTimerChangePeriod() API function example usage scenario.

Parameters

- **xTimer** –The handle of the timer being deleted.
- **xTicksToWait** –Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when xTimerDelete() was called. xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started.

Returns pdFAIL will be returned if the delete command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerReset(xTimer, xTicksToWait)

BaseType_t xTimerReset(TimerHandle_t xTimer, TickType_t xTicksToWait);

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerReset() re-starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerReset() will cause the timer to re-evaluate its expiry time so that it is relative to when xTimerReset() was called. If the timer was in the dormant state then xTimerReset() has equivalent functionality to the xTimerStart() API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called, where 'n' is the timers defined period.

It is valid to call xTimerReset() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerReset() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerReset() to be available.

Example usage:

```
* // When a key is pressed, an LCD back-light is switched on.  If 5 seconds␣
↪pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer.
*
* TimerHandle_t xBacklightTimer = NULL;
*
* // The callback function assigned to the one-shot timer.  In this case the
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press event handler.
* void vKeyPressEventHandler( char cKey )
* {
*     // Ensure the LCD back-light is on, then reset the timer that is
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  Wait 10 ticks for the command to be successfully sent
*     // if it cannot be sent immediately.
```

(continues on next page)

(continued from previous page)

```
*     vSetBacklightState( BACKLIGHT_ON );
*     if( xTimerReset( xBacklightTimer, 100 ) != pdPASS )
*     {
*         // The reset command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
* }
*
* void main( void )
* {
* int32_t x;
*
*     // Create then start the one-shot timer that is responsible for turning
*     // the back-light off if no keys are pressed within a 5 second period.
*     xBacklightTimer = xTimerCreate( "BacklightTimer",           // Just a␣
↪text name, not used by the kernel.
*                                     ( 5000 / portTICK_PERIOD_MS), // The␣
↪timer period in ticks.
*                                     pdFALSE,                    // The timer␣
↪is a one-shot timer.
*                                     0,                          // The id is␣
↪not used by the callback so can take any value.
*                                     vBacklightTimerCallback     // The␣
↪callback function that switches the LCD back-light off.
*                                   );
*
*     if( xBacklightTimer == NULL )
*     {
*         // The timer was not created.
*     }
*     else
*     {
*         // Start the timer.  No block time is specified, and even if one was
*         // it would be ignored because the scheduler has not yet been
*         // started.
*         if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
*         {
*             // The timer could not be set into the Active state.
*         }
*     }
*
*     // ...
*     // Create tasks here.
*     // ...
*
*     // Starting the scheduler will start the timer running as it has already
*     // been set into the active state.
*     vTaskStartScheduler();
*
*     // Should not reach here.
*     for( ;; );
* }
*
```

Parameters

- **xTimer** –The handle of the timer being reset/started/restarted.
- **xTicksToWait** –Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called. xTicksToWait is

ignored if xTimerReset() is called before the scheduler is started.

Returns pdFAIL will be returned if the reset command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerStartFromISR(xTimer, pxHigherPriorityTaskWoken)

BaseType_t xTimerStartFromISR(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);

A version of xTimerStart() that can be called from an interrupt service routine.

Example usage:

```
* // This scenario assumes xBacklightTimer has already been created.  When a
* // key is pressed, an LCD back-light is switched on.  If 5 seconds pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer, and unlike the example given for
* // the xTimerReset() function, the key press event handler is an interrupt
* // service routine.
*
* // The callback function assigned to the one-shot timer.  In this case the
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press interrupt service routine.
* void vKeyPressEventInterruptHandler( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // Ensure the LCD back-light is on, then restart the timer that is
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  This is an interrupt service routine so can only
*     // call FreeRTOS API functions that end in "FromISR".
*     vSetBacklightState( BACKLIGHT_ON );
*
*     // xTimerStartFromISR() or xTimerResetFromISR() could be called here
*     // as both cause the timer to re-calculate its expiry time.
*     // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
*     // declared (in this function).
*     if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=␣
↪pdPASS )
*     {
*         // The start command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
```

(continues on next page)

(continued from previous page)

```
*      if( xHigherPriorityTaskWoken != pdFALSE )
*      {
*          // Call the interrupt safe yield function here (actual function
*          // depends on the FreeRTOS port being used).
*      }
* }
*
```

Parameters

- **xTimer** –The handle of the timer being started/restarted.
- **pxHigherPriorityTaskWoken** –The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStartFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStartFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns pdFAIL will be returned if the start command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStartFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerStopFromISR (xTimer, pxHigherPriorityTaskWoken)

BaseType_t xTimerStopFromISR(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);

A version of xTimerStop() that can be called from an interrupt service routine.

Example usage:

```
* // This scenario assumes xTimer has already been created and started.  When
* // an interrupt occurs, the timer should be simply stopped.
*
* // The interrupt service routine that stops the timer.
* void vAnExampleInterruptServiceRoutine( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*      // The interrupt has occurred - simply stop the timer.
*      // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
*      // (within this function).  As this is an interrupt service routine, only
*      // FreeRTOS API functions that end in "FromISR" can be used.
*      if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
*      {
*          // The stop command was not executed successfully.  Take appropriate
*          // action here.
*      }
*
*      // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*      // should be performed.  The syntax required to perform a context switch
*      // from inside an ISR varies from port to port, and from compiler to
*      // compiler.  Inspect the demos for the port you are using to find the
*      // actual syntax required.
*      if( xHigherPriorityTaskWoken != pdFALSE )
```

(continues on next page)

(continued from previous page)

```
*      {
*          // Call the interrupt safe yield function here (actual function
*          // depends on the FreeRTOS port being used).
*      }
* }
*
```

Parameters

- **xTimer** –The handle of the timer being stopped.
- **pxHigherPriorityTaskWoken** –The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStopFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStopFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns pdFAIL will be returned if the stop command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerChangePeriodFromISR(xTimer, xNewPeriod, pxHigherPriorityTaskWoken)

BaseType_t xTimerChangePeriodFromISR(TimerHandle_t xTimer, TickType_t xNewPeriod, BaseType_t *pxHigherPriorityTaskWoken);

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Example usage:

```
* // This scenario assumes xTimer has already been created and started.  When
* // an interrupt occurs, the period of xTimer should be changed to 500ms.
*
* // The interrupt service routine that changes the period of xTimer.
* void vAnExampleInterruptServiceRoutine( void )
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // The interrupt has occurred - change the period of xTimer to 500ms.
*     // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
*     // (within this function).  As this is an interrupt service routine, only
*     // FreeRTOS API functions that end in "FromISR" can be used.
*     if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) !=␣
↪pdPASS )
*     {
*         // The command to change the timers period was not executed
*         // successfully.  Take appropriate action here.
*     }
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
```

(continues on next page)

(continued from previous page)

```
*      {
*          // Call the interrupt safe yield function here (actual function
*          // depends on the FreeRTOS port being used).
*      }
* }
*
```

Parameters

- **xTimer** –The handle of the timer that is having its period changed.
- **xNewPeriod** –The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- **pxHigherPriorityTaskWoken** –The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerChangePeriodFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/ daemon task out of the Blocked state. If calling xTimerChangePeriodFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns pdFAIL will be returned if the command to change the timers period could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

xTimerResetFromISR(xTimer, pxHigherPriorityTaskWoken)

BaseType_t xTimerResetFromISR(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);

A version of xTimerReset() that can be called from an interrupt service routine.

Example usage:

```
* // This scenario assumes xBacklightTimer has already been created.  When a
* // key is pressed, an LCD back-light is switched on.  If 5 seconds pass
* // without a key being pressed, then the LCD back-light is switched off.  In
* // this case, the timer is a one-shot timer, and unlike the example given for
* // the xTimerReset() function, the key press event handler is an interrupt
* // service routine.
*
* // The callback function assigned to the one-shot timer.  In this case the
* // parameter is not used.
* void vBacklightTimerCallback( TimerHandle_t pxTimer )
* {
*     // The timer expired, therefore 5 seconds must have passed since a key
*     // was pressed.  Switch off the LCD back-light.
*     vSetBacklightState( BACKLIGHT_OFF );
* }
*
* // The key press interrupt service routine.
* void vKeyPressEventInterruptHandler( void )
```

(continues on next page)

(continued from previous page)

```
* {
* BaseType_t xHigherPriorityTaskWoken = pdFALSE;
*
*     // Ensure the LCD back-light is on, then reset the timer that is
*     // responsible for turning the back-light off after 5 seconds of
*     // key inactivity.  This is an interrupt service routine so can only
*     // call FreeRTOS API functions that end in "FromISR".
*     vSetBacklightState( BACKLIGHT_ON );
*
*     // xTimerStartFromISR() or xTimerResetFromISR() could be called here
*     // as both cause the timer to re-calculate its expiry time.
*     // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
*     // declared (in this function).
*     if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=␣
↪pdPASS )
*     {
*         // The reset command was not executed successfully.  Take appropriate
*         // action here.
*     }
*
*     // Perform the rest of the key processing here.
*
*     // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
*     // should be performed.  The syntax required to perform a context switch
*     // from inside an ISR varies from port to port, and from compiler to
*     // compiler.  Inspect the demos for the port you are using to find the
*     // actual syntax required.
*     if( xHigherPriorityTaskWoken != pdFALSE )
*     {
*         // Call the interrupt safe yield function here (actual function
*         // depends on the FreeRTOS port being used).
*     }
* }
*
```

Parameters

- **xTimer** –The handle of the timer that is to be started, reset, or restarted.
- **pxHigherPriorityTaskWoken** –The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerResetFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerResetFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerReset-FromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Type Definitions

typedef struct tmrTimerControl ***TimerHandle_t**

typedef void (***TimerCallbackFunction_t**)(*TimerHandle_t* xTimer)

typedef void (*`PendedFunction_t`)(void*, uint32_t)

Event Group API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/event_groups.h

Functions

EventGroupHandle_t **`xEventGroupCreate`**(void)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGroupCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see https://www.FreeRTOS.org/a00111.html). If an event group is created using xEventGroupCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

Example usage:

```
// Declare a variable to hold the created event group.
EventGroupHandle_t xCreatedEventGroup;

// Attempt to create the event group.
xCreatedEventGroup = xEventGroupCreate();

// Was the event group created successfully?
if( xCreatedEventGroup == NULL )
{
    // The event group was not created because there was insufficient
    // FreeRTOS heap available.
}
else
{
    // The event group was created.
}
```

Returns If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned. See https://www.FreeRTOS.org/a00111.html

EventGroupHandle_t **`xEventGroupCreateStatic`**(StaticEventGroup_t *pxEventGroupBuffer)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGroupCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see https://www.FreeRTOS.org/a00111.html). If an event group is created using xEventGroupCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

Example usage:

```
// StaticEventGroup_t is a publicly accessible structure that has the same
// size and alignment requirements as the real event group structure.  It is
// provided as a mechanism for applications to know the size of the event
// group (which is dependent on the architecture and configuration file
// settings) without breaking the strict data hiding policy by exposing the
// real event group internals.  This StaticEventGroup_t variable is passed
// into the xSemaphoreCreateEventGroupStatic() function and is used to store
// the event group's data structures
StaticEventGroup_t xEventGroupBuffer;

// Create the event group without dynamically allocating any memory.
xEventGroup = xEventGroupCreateStatic( &xEventGroupBuffer );
```

Parameters **`pxEventGroupBuffer`** –pxEventGroupBuffer must point to a variable of type StaticEventGroup_t, which will be then be used to hold the event group' s data structures, removing the need for the memory to be allocated dynamically.

Returns If the event group was created then a handle to the event group is returned. If pxEventGroupBuffer was NULL then NULL is returned.

EventBits_t **`xEventGroupWaitBits`**(*EventGroupHandle_t* xEventGroup, const *EventBits_t* uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait)

[Potentially] block to wait for one or more bits to be set within a previously created event group.

This function cannot be called from an interrupt.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    // Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    // the event group.  Clear the bits before exiting.
    uxBits = xEventGroupWaitBits(
                xEventGroup,    // The event group being tested.
                BIT_0 | BIT_4,  // The bits within the event group to wait␣
↪for.
                pdTRUE,         // BIT_0 and BIT_4 should be cleared before␣
↪returning.
                pdFALSE,        // Don't wait for both bits, either bit will␣
↪do.
                xTicksToWait ); // Wait a maximum of 100ms for either bit to␣
↪be set.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
```

(continues on next page)

(continued from previous page)

```
    {
        // xEventGroupWaitBits() returned because both bits were set.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_0 was set.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_4 was set.
    }
    else
    {
        // xEventGroupWaitBits() returned because xTicksToWait ticks passed
        // without either BIT_0 or BIT_4 becoming set.
    }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().
- **uxBitsToWaitFor** –A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc.
- **xClearOnExit** –If xClearOnExit is set to pdTRUE then any bits within uxBitsToWaitFor that are set within the event group will be cleared before xEventGroupWaitBits() returns if the wait condition was met (if the function returns for a reason other than a timeout). If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns.
- **xWaitForAllBits** –If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either all the bits in uxBitsToWaitFor are set or the specified block time expires. If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when any one of the bits set in uxBitsToWaitFor is set or the specified block time expires. The block time is specified by the xTicksToWait parameter.
- **xTicksToWait** –The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Returns The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that xClearOnExit parameter was set to pdTRUE.

EventBits_t **xEventGroupClearBits**(*EventGroupHandle_t* xEventGroup, const *EventBits_t* uxBitsToClear)

Clear bits within an event group. This function cannot be called from an interrupt.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    // Clear bit 0 and bit 4 in xEventGroup.
```

(continues on next page)

(continued from previous page)

```
    uxBits = xEventGroupClearBits(
                            xEventGroup,    // The event group being updated.
                            BIT_0 | BIT_4 );// The bits being cleared.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 were set before xEventGroupClearBits() was
        // called.  Both will now be clear (not set).
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else
    {
        // Neither bit 0 nor bit 4 were set in the first place.
    }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are to be cleared.
- **uxBitsToClear** –A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09.

Returns The value of the event group before the specified bits were cleared.

EventBits_t **xEventGroupSetBits**(*EventGroupHandle_t* xEventGroup, const *EventBits_t* uxBitsToSet)

Set bits within an event group. This function cannot be called from an interrupt. xEventGroupSetBitsFromISR() is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    // Set bit 0 and bit 4 in xEventGroup.
    uxBits = xEventGroupSetBits(
                        xEventGroup,    // The event group being updated.
                        BIT_0 | BIT_4 );// The bits being set.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 remained set when the function returned.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 remained set when the function returned, but bit 4 was
```

(continues on next page)

(continued from previous page)

```
            // cleared.  It might be that bit 4 was cleared automatically as a
            // task that was waiting for bit 4 was removed from the Blocked
            // state.
        }
        else if( ( uxBits & BIT_4 ) != 0 )
        {
            // Bit 4 remained set when the function returned, but bit 0 was
            // cleared.  It might be that bit 0 was cleared automatically as a
            // task that was waiting for bit 0 was removed from the Blocked
            // state.
        }
        else
        {
            // Neither bit 0 nor bit 4 remained set.  It might be that a task
            // was waiting for both of the bits to be set, and the bits were
            // cleared as the task left the Blocked state.
        }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are to be set.
- **uxBitsToSet** –A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09.

Returns The value of the event group at the time the call to xEventGroupSetBits() returns. There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared. First, if setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will be cleared automatically (see the xClearBitOnExit parameter of xEventGroupWaitBits()). Second, any unblocked (or otherwise Ready state) task that has a priority above that of the task that called xEventGroupSetBits() will execute and may change the event group value before the call to xEventGroupSetBits() returns.

EventBits_t **xEventGroupSync**(*EventGroupHandle_t* xEventGroup, const *EventBits_t* uxBitsToSet, const *EventBits_t* uxBitsToWaitFor, TickType_t xTicksToWait)

Atomically set bits within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronise multiple tasks, where each task has to wait for the other tasks to reach a synchronisation point before proceeding.

This function cannot be used from an interrupt.

The function will return before its block time expires if the bits specified by the uxBitsToWait parameter are set, or become set within that time. In this case all the bits specified by uxBitsToWait will be automatically cleared before the function returns.

Example usage:

```
// Bits used by the three tasks.
#define TASK_0_BIT      ( 1 << 0 )
#define TASK_1_BIT      ( 1 << 1 )
#define TASK_2_BIT      ( 1 << 2 )

#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

// Use an event group to synchronise three tasks.  It is assumed this event
// group has already been created elsewhere.
EventGroupHandle_t xEventBits;

void vTask0( void *pvParameters )
{
```

(continues on next page)

(continued from previous page)

```
EventBits_t uxReturn;
TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

  for( ;; )
  {
     // Perform task functionality here.

     // Set bit 0 in the event flag to note this task has reached the
     // sync point.  The other two tasks will set the other two bits defined
     // by ALL_SYNC_BITS.  All three tasks have reached the synchronisation
     // point when all the ALL_SYNC_BITS are set.  Wait a maximum of 100ms
     // for this to happen.
     uxReturn = xEventGroupSync( xEventBits, TASK_0_BIT, ALL_SYNC_BITS,␣
↪xTicksToWait );

     if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
     {
         // All three tasks reached the synchronisation point before the call
         // to xEventGroupSync() timed out.
     }
 }
}

void vTask1( void *pvParameters )
{
  for( ;; )
  {
     // Perform task functionality here.

     // Set bit 1 in the event flag to note this task has reached the
     // synchronisation point.  The other two tasks will set the other two
     // bits defined by ALL_SYNC_BITS.  All three tasks have reached the
     // synchronisation point when all the ALL_SYNC_BITS are set.  Wait
     // indefinitely for this to happen.
     xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );

     // xEventGroupSync() was called with an indefinite block time, so
     // this task will only reach here if the synchronisation was made by all
     // three tasks, so there is no need to test the return value.
  }
}

void vTask2( void *pvParameters )
{
  for( ;; )
  {
     // Perform task functionality here.

     // Set bit 2 in the event flag to note this task has reached the
     // synchronisation point.  The other two tasks will set the other two
     // bits defined by ALL_SYNC_BITS.  All three tasks have reached the
     // synchronisation point when all the ALL_SYNC_BITS are set.  Wait
     // indefinitely for this to happen.
     xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS, portMAX_DELAY );

     // xEventGroupSync() was called with an indefinite block time, so
     // this task will only reach here if the synchronisation was made by all
     // three tasks, so there is no need to test the return value.
 }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().
- **uxBitsToSet** –The bits to set in the event group before determining if, and possibly waiting for, all the bits specified by the uxBitsToWait parameter are set.
- **uxBitsToWaitFor** –A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and bit 1 and bit 2 set uxBitsToWaitFor to 0x07. Etc.
- **xTicksToWait** –The maximum amount of time (specified in 'ticks') to wait for all of the bits specified by uxBitsToWaitFor to become set.

Returns The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupSync() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupSync() returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

EventBits_t **xEventGroupGetBitsFromISR**(*EventGroupHandle_t* xEventGroup)

A version of xEventGroupGetBits() that can be called from an ISR.

Parameters **xEventGroup** –The event group being queried.
Returns The event group bits at the time xEventGroupGetBitsFromISR() was called.

void **vEventGroupDelete**(*EventGroupHandle_t* xEventGroup)

Delete an event group that was previously created by a call to xEventGroupCreate(). Tasks that are blocked on the event group will be unblocked and obtain 0 as the event group's value.

Parameters **xEventGroup** –The event group being deleted.

Macros

xEventGroupClearBitsFromISR(xEventGroup, uxBitsToClear)

A version of xEventGroupClearBits() that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed while interrupts are disabled, so protects event groups that are accessed from tasks by suspending the scheduler rather than disabling interrupts. As a result event groups cannot be accessed directly from an interrupt service routine. Therefore xEventGroupClearBitsFromISR() sends a message to the timer task to have the clear operation performed in the context of the timer task.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
     // Clear bit 0 and bit 4 in xEventGroup.
     xResult = xEventGroupClearBitsFromISR(
                         xEventGroup,     // The event group being updated.
                         BIT_0 | BIT_4 ); // The bits being set.

     if( xResult == pdPASS )
     {
         // The message was posted successfully.
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are to be cleared.
- **uxBitsToClear** –A bitwise value that indicates the bit or bits to clear. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09.

Returns If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

xEventGroupSetBitsFromISR(xEventGroup, uxBitsToSet, pxHigherPriorityTaskWoken)

A version of xEventGroupSetBits() that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitsFromISR() sends a message to the timer task to have the set operation performed in the context of the timer task - where a scheduler lock is used in place of a critical section.

Example usage:

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken, xResult;

     // xHigherPriorityTaskWoken must be initialised to pdFALSE.
     xHigherPriorityTaskWoken = pdFALSE;

     // Set bit 0 and bit 4 in xEventGroup.
     xResult = xEventGroupSetBitsFromISR(
                         xEventGroup,    // The event group being updated.
                         BIT_0 | BIT_4   // The bits being set.
                         &xHigherPriorityTaskWoken );

     // Was the message posted successfully?
     if( xResult == pdPASS )
     {
          // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
          // switch should be requested.  The macro used is port specific and
          // will be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
          // refer to the documentation page for the port being used.
          portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
     }
}
```

Parameters

- **xEventGroup** –The event group in which the bits are to be set.
- **uxBitsToSet** –A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09.

- **pxHigherPriorityTaskWoken** –As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

xEventGroupGetBits (xEventGroup)

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

Parameters

- **xEventGroup** –The event group being queried.

Returns The event group bits at the time xEventGroupGetBits() was called.

Type Definitions

typedef struct EventGroupDef_t ***EventGroupHandle_t**

typedef TickType_t **EventBits_t**

Stream Buffer API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/stream_buffer.h

Functions

size_t **xStreamBufferSend** (*StreamBufferHandle_t* xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait)

Sends bytes to a stream buffer. The bytes are copied into the stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

Example use:

```
void vAFunction( StreamBufferHandle_t xStreamBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
const TickType_t x100ms = pdMS_TO_TICKS( 100 );
```

(continues on next page)

(continued from previous page)

```
// Send an array to the stream buffer, blocking for a maximum of 100ms to
// wait for enough space to be available in the stream buffer.
xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) ucArrayToSend,␣
↪sizeof( ucArrayToSend ), x100ms );

if( xBytesSent != sizeof( ucArrayToSend ) )
{
    // The call to xStreamBufferSend() times out before there was enough
    // space in the buffer for the data to be written, but it did
    // successfully write xBytesSent bytes.
}

// Send the string to the stream buffer.  Return immediately if there is not
// enough space in the buffer.
xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) pcStringToSend,␣
↪strlen( pcStringToSend ), 0 );

if( xBytesSent != strlen( pcStringToSend ) )
{
    // The entire string could not be added to the stream buffer because
    // there was not enough free space in the buffer, but xBytesSent bytes
    // were sent.  Could try again to send the remaining bytes.
}
}
```

Parameters

- **xStreamBuffer** –The handle of the stream buffer to which a stream is being sent.
- **pvTxData** –A pointer to the buffer that holds the bytes to be copied into the stream buffer.
- **xDataLengthBytes** –The maximum number of bytes to copy from pvTxData into the stream buffer.
- **xTicksToWait** –The maximum amount of time the task should remain in the Blocked state to wait for enough space to become available in the stream buffer, should the stream buffer contain too little space to hold the another xDataLengthBytes bytes. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible. A task does not use any CPU time when it is in the blocked state.

Returns The number of bytes written to the stream buffer. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible.

size_t **xStreamBufferSendFromISR**(*StreamBufferHandle_t* xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, BaseType_t *const pxHigherPriorityTaskWoken)

Interrupt safe version of the API function that sends a stream of bytes to the stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write

to a stream buffer from an interrupt service routine (ISR).

Example use:

```
// A stream buffer that has already been created.
StreamBufferHandle_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

 // Attempt to send the string to the stream buffer.
 xBytesSent = xStreamBufferSendFromISR( xStreamBuffer,
                                        ( void * ) pcStringToSend,
                                        strlen( pcStringToSend ),
                                        &xHigherPriorityTaskWoken );

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // There was not enough free space in the stream buffer for the entire
     // string to be written, ut xBytesSent bytes were written.
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xStreamBufferSendFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Parameters

- **xStreamBuffer** –The handle of the stream buffer to which a stream is being sent.
- **pvTxData** –A pointer to the data that is to be copied into the stream buffer.
- **xDataLengthBytes** –The maximum number of bytes to copy from pvTxData into the stream buffer.
- **pxHigherPriorityTaskWoken** –It is possible that a stream buffer will have a task blocked on it waiting for data. Calling xStreamBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xStreamBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the example code below for an example.

Returns The number of bytes actually written to the stream buffer, which will be less than xDataLengthBytes if the stream buffer didn't have enough free space for all the bytes to be written.

size_t **xStreamBufferReceive**(*StreamBufferHandle_t* xStreamBuffer, void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait)

Receives bytes from a stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferReceive() to read from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read from a stream buffer from an interrupt service routine (ISR).

Example use:

```
void vAFunction( StreamBuffer_t xStreamBuffer )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

 // Receive up to another sizeof( ucRxData ) bytes from the stream buffer.
 // Wait in the Blocked state (so not using any CPU processing time) for a
 // maximum of 100ms for the full sizeof( ucRxData ) number of bytes to be
 // available.
 xReceivedBytes = xStreamBufferReceive( xStreamBuffer,
                                        ( void * ) ucRxData,
                                        sizeof( ucRxData ),
                                        xBlockTime );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains another xRecievedBytes bytes of data, which can
     // be processed here....
 }
}
```

Parameters

- **xStreamBuffer** –The handle of the stream buffer from which bytes are to be received.
- **pvRxData** –A pointer to the buffer into which the received bytes will be copied.
- **xBufferLengthBytes** –The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
- **xTicksToWait** –The maximum amount of time the task should remain in the Blocked state to wait for data to become available if the stream buffer is empty. xStreamBufferReceive() will return immediately if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. A task does not use any CPU time when it is in the Blocked state.

Returns The number of bytes actually read from the stream buffer, which will be less than xBufferLengthBytes if the call to xStreamBufferReceive() timed out before xBufferLengthBytes were available.

size_t **xStreamBufferReceiveFromISR**(*StreamBufferHandle_t* xStreamBuffer, void *pvRxData, size_t xBufferLengthBytes, BaseType_t *const pxHigherPriorityTaskWoken)

An interrupt safe version of the API function that receives bytes from a stream buffer.

Use xStreamBufferReceive() to read bytes from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read bytes from a stream buffer from an interrupt service routine (ISR).

Example use:

```
// A stream buffer that has already been created.
StreamBuffer_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;  // Initialised to pdFALSE.

 // Receive the next stream from the stream buffer.
 xReceivedBytes = xStreamBufferReceiveFromISR( xStreamBuffer,
                                               ( void * ) ucRxData,
                                               sizeof( ucRxData ),
                                               &xHigherPriorityTaskWoken );

 if( xReceivedBytes > 0 )
 {
     // ucRxData contains xReceivedBytes read from the stream buffer.
     // Process the stream here....
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xStreamBufferReceiveFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Parameters

- **xStreamBuffer** –The handle of the stream buffer from which a stream is being received.
- **pvRxData** –A pointer to the buffer into which the received bytes are copied.
- **xBufferLengthBytes** –The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
- **pxHigherPriorityTaskWoken** –It is possible that a stream buffer will have a task blocked on it waiting for space to become available. Calling xStreamBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xStreamBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferReceiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Returns The number of bytes read from the stream buffer, if any.

void **vStreamBufferDelete**(*StreamBufferHandle_t* xStreamBuffer)

Deletes a stream buffer that was previously created using a call to xStreamBufferCreate() or xStreamBuffer-

CreateStatic(). If the stream buffer was created using dynamic memory (that is, by xStreamBufferCreate()), then the allocated memory is freed.

A stream buffer handle must not be used after the stream buffer has been deleted.

Parameters **`xStreamBuffer`** –The handle of the stream buffer to be deleted.

BaseType_t **`xStreamBufferIsFull`** (*StreamBufferHandle_t* xStreamBuffer)

Queries a stream buffer to see if it is full. A stream buffer is full if it does not have any free space, and therefore cannot accept any more data.

Parameters **`xStreamBuffer`** –The handle of the stream buffer being queried.
Returns If the stream buffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t **`xStreamBufferIsEmpty`** (*StreamBufferHandle_t* xStreamBuffer)

Queries a stream buffer to see if it is empty. A stream buffer is empty if it does not contain any data.

Parameters **`xStreamBuffer`** –The handle of the stream buffer being queried.
Returns If the stream buffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t **`xStreamBufferReset`** (*StreamBufferHandle_t* xStreamBuffer)

Resets a stream buffer to its initial, empty, state. Any data that was in the stream buffer is discarded. A stream buffer can only be reset if there are no tasks blocked waiting to either send to or receive from the stream buffer.

Parameters **`xStreamBuffer`** –The handle of the stream buffer being reset.
Returns If the stream buffer is reset then pdPASS is returned. If there was a task blocked waiting to send to or read from the stream buffer then the stream buffer is not reset and pdFAIL is returned.

size_t **`xStreamBufferSpacesAvailable`** (*StreamBufferHandle_t* xStreamBuffer)

Queries a stream buffer to see how much free space it contains, which is equal to the amount of data that can be sent to the stream buffer before it is full.

Parameters **`xStreamBuffer`** –The handle of the stream buffer being queried.
Returns The number of bytes that can be written to the stream buffer before the stream buffer would be full.

size_t **`xStreamBufferBytesAvailable`** (*StreamBufferHandle_t* xStreamBuffer)

Queries a stream buffer to see how much data it contains, which is equal to the number of bytes that can be read from the stream buffer before the stream buffer would be empty.

Parameters **`xStreamBuffer`** –The handle of the stream buffer being queried.
Returns The number of bytes that can be read from the stream buffer before the stream buffer would be empty.

BaseType_t **`xStreamBufferSetTriggerLevel`** (*StreamBufferHandle_t* xStreamBuffer, size_t xTriggerLevel)

A stream buffer's trigger level is the number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

A trigger level is set when the stream buffer is created, and can be modified using xStreamBufferSetTriggerLevel().

Parameters

- **`xStreamBuffer`** –The handle of the stream buffer being updated.
- **`xTriggerLevel`** –The new trigger level for the stream buffer.

Returns If xTriggerLevel was less than or equal to the stream buffer's length then the trigger level will be updated and pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t **xStreamBufferSendCompletedFromISR**(*StreamBufferHandle_t* xStreamBuffer, BaseType_t *pxHigherPriorityTaskWoken)

For advanced users only.

The sbSEND_COMPLETED() macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbSEND_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferSendCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbSEND_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

Parameters

- **xStreamBuffer** –The handle of the stream buffer to which data was written.
- **pxHigherPriorityTaskWoken** –*pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferSendCompletedFromISR(). If calling xStreamBufferSendCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

Returns If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t **xStreamBufferReceiveCompletedFromISR**(*StreamBufferHandle_t* xStreamBuffer, BaseType_t *pxHigherPriorityTaskWoken)

For advanced users only.

The sbRECEIVE_COMPLETED() macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbRECEIVE_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferReceiveCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbRECEIVE_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

Parameters

- **xStreamBuffer** –The handle of the stream buffer from which data was read.
- **pxHigherPriorityTaskWoken** –*pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferReceiveCompletedFromISR(). If calling xStreamBufferReceiveCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

Returns If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

Macros

xStreamBufferCreate(xBufferSizeBytes, xTriggerLevelBytes)

Creates a new stream buffer using dynamically allocated memory. See xStreamBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xStreamBufferCreate() to be available.

Example use:

```
void vAFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;

 // Create a stream buffer that can hold 100 bytes.  The memory used to hold
 // both the stream buffer structure and the data in the stream buffer is
 // allocated dynamically.
 xStreamBuffer = xStreamBufferCreate( xStreamBufferSizeBytes, xTriggerLevel );

 if( xStreamBuffer == NULL )
 {
     // There was not enough heap memory space available to create the
     // stream buffer.
 }
 else
 {
     // The stream buffer was created successfully and can now be used.
 }
}
```

Parameters

- **xBufferSizeBytes** –The total number of bytes the stream buffer will be able to hold at any one time.
- **xTriggerLevelBytes** –The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

Returns If NULL is returned, then the stream buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the stream buffer data structures and storage area. A non-NULL value being returned indicates that the stream buffer has been created successfully - the returned value should be stored as the handle to the created stream buffer.

xStreamBufferCreateStatic(xBufferSizeBytes, xTriggerLevelBytes, pucStreamBufferStorageArea, pxStaticStreamBuffer)

Creates a new stream buffer using statically allocated memory. See xStreamBufferCreate() for a version that uses dynamically allocated memory.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for xStreamBufferCreateStatic() to be available.

Example use:

```
// Used to dimension the array used to hold the streams.  The available space
// will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

// Defines the memory that will actually hold the streams within the stream
// buffer.
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

// The variable used to hold the stream buffer structure.
```

(continues on next page)

(continued from previous page)

```
StaticStreamBuffer_t xStreamBufferStruct;

void MyFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xTriggerLevel = 1;

 xStreamBuffer = xStreamBufferCreateStatic( sizeof( ucBufferStorage ),
                                            xTriggerLevel,
                                            ucBufferStorage,
                                            &xStreamBufferStruct );

 // As neither the pucStreamBufferStorageArea or pxStaticStreamBuffer
 // parameters were NULL, xStreamBuffer will not be NULL, and can be used to
 // reference the created stream buffer in other stream buffer API calls.

 // Other code that uses the stream buffer can go here.
}
```

Parameters

- **xBufferSizeBytes** –The size, in bytes, of the buffer pointed to by the pucStreamBufferStorageArea parameter.
- **xTriggerLevelBytes** –The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.
- **pucStreamBufferStorageArea** –Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which streams are copied when they are written to the stream buffer.
- **pxStaticStreamBuffer** –Must point to a variable of type StaticStreamBuffer_t, which will be used to hold the stream buffer's data structure.

Returns If the stream buffer is created successfully then a handle to the created stream buffer is returned. If either pucStreamBufferStorageArea or pxStaticstreamBuffer are NULL then NULL is returned.

Type Definitions

typedef struct StreamBufferDef_t ***StreamBufferHandle_t**

Message Buffer API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/message_buffer.h

Macros

xMessageBufferCreate(xBufferSizeBytes)

Creates a new message buffer using dynamically allocated memory. See xMessageBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xMessageBufferCreate() to be available.

Example use:

```
void vAFunction( void )
{
MessageBufferHandle_t xMessageBuffer;
const size_t xMessageBufferSizeBytes = 100;

 // Create a message buffer that can hold 100 bytes.  The memory used to hold
 // both the message buffer structure and the messages themselves is allocated
 // dynamically.  Each message added to the buffer consumes an additional 4
 // bytes which are used to hold the lengh of the message.
 xMessageBuffer = xMessageBufferCreate( xMessageBufferSizeBytes );

 if( xMessageBuffer == NULL )
 {
     // There was not enough heap memory space available to create the
     // message buffer.
 }
 else
 {
     // The message buffer was created successfully and can now be used.
 }
```

Parameters

- **xBufferSizeBytes** –The total number of bytes (not messages) the message buffer will be able to hold at any one time. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message' s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architectures a 10 byte message will take up 14 bytes of message buffer space.

Returns If NULL is returned, then the message buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the message buffer data structures and storage area. A non-NULL value being returned indicates that the message buffer has been created successfully - the returned value should be stored as the handle to the created message buffer.

xMessageBufferCreateStatic(xBufferSizeBytes, pucMessageBufferStorageArea, pxStaticMessageBuffer)

Creates a new message buffer using statically allocated memory. See xMessageBufferCreate() for a version that uses dynamically allocated memory.

Example use:

```
// Used to dimension the array used to hold the messages.  The available space
// will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

// Defines the memory that will actually hold the messages within the message
// buffer.
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

// The variable used to hold the message buffer structure.
```

(continues on next page)

(continued from previous page)

```
StaticMessageBuffer_t xMessageBufferStruct;

void MyFunction( void )
{
MessageBufferHandle_t xMessageBuffer;

 xMessageBuffer = xMessageBufferCreateStatic( sizeof( ucBufferStorage ),
                                              ucBufferStorage,
                                              &xMessageBufferStruct );

 // As neither the pucMessageBufferStorageArea or pxStaticMessageBuffer
 // parameters were NULL, xMessageBuffer will not be NULL, and can be used to
 // reference the created message buffer in other message buffer API calls.

 // Other code that uses the message buffer can go here.
}
```

Parameters

- **xBufferSizeBytes** –The size, in bytes, of the buffer pointed to by the pucMessageBufferStorageArea parameter. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture a 10 byte message will take up 14 bytes of message buffer space. The maximum number of bytes that can be stored in the message buffer is actually (xBufferSizeBytes - 1).
- **pucMessageBufferStorageArea** –Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which messages are copied when they are written to the message buffer.
- **pxStaticMessageBuffer** –Must point to a variable of type StaticMessageBuffer_t, which will be used to hold the message buffer's data structure.

Returns If the message buffer is created successfully then a handle to the created message buffer is returned. If either pucMessageBufferStorageArea or pxStaticmessageBuffer are NULL then NULL is returned.

xMessageBufferSend(xMessageBuffer, pvTxData, xDataLengthBytes, xTicksToWait)

Sends a discrete message to the message buffer. The message can be any length that fits within the buffer's free space, and is copied into the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

Example use:

```
void vAFunction( MessageBufferHandle_t xMessageBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
```

(continues on next page)

(continued from previous page)

```
const TickType_t x100ms = pdMS_TO_TICKS( 100 );

 // Send an array to the message buffer, blocking for a maximum of 100ms to
 // wait for enough space to be available in the message buffer.
 xBytesSent = xMessageBufferSend( xMessageBuffer, ( void * ) ucArrayToSend,␣
↪sizeof( ucArrayToSend ), x100ms );

 if( xBytesSent != sizeof( ucArrayToSend ) )
 {
     // The call to xMessageBufferSend() times out before there was enough
     // space in the buffer for the data to be written.
 }

 // Send the string to the message buffer.  Return immediately if there is
 // not enough space in the buffer.
 xBytesSent = xMessageBufferSend( xMessageBuffer, ( void * ) pcStringToSend,␣
↪strlen( pcStringToSend ), 0 );

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // The string could not be added to the message buffer because there was
     // not enough free space in the buffer.
 }
}
```

Parameters

- **xMessageBuffer** –The handle of the message buffer to which a message is being sent.
- **pvTxData** –A pointer to the message that is to be copied into the message buffer.
- **xDataLengthBytes** –The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message' s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length).
- **xTicksToWait** –The maximum amount of time the calling task should remain in the Blocked state to wait for enough space to become available in the message buffer, should the message buffer have insufficient space when xMessageBufferSend() is called. The calling task will never block if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Returns The number of bytes written to the message buffer. If the call to xMessageBufferSend() times out before there was enough space to write the message into the message buffer then zero is returned. If the call did not time out then xDataLengthBytes is returned.

xMessageBufferSendFromISR(xMessageBuffer, pvTxData, xDataLengthBytes, pxHigherPriorityTaskWoken)

Interrupt safe version of the API function that sends a discrete message to the message buffer. The message can be any length that fits within the buffer' s free space, and is copied into the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers

then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

Example use:

```
// A message buffer that has already been created.
MessageBufferHandle_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

 // Attempt to send the string to the message buffer.
 xBytesSent = xMessageBufferSendFromISR( xMessageBuffer,
                                         ( void * ) pcStringToSend,
                                         strlen( pcStringToSend ),
                                         &xHigherPriorityTaskWoken );

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // The string could not be added to the message buffer because there was
     // not enough free space in the buffer.
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xMessageBufferSendFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Parameters

- **xMessageBuffer** –The handle of the message buffer to which a message is being sent.
- **pvTxData** –A pointer to the message that is to be copied into the message buffer.
- **xDataLengthBytes** –The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message' s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length).
- **pxHigherPriorityTaskWoken** –It is possible that a message buffer will have a task blocked on it waiting for data. Calling xMessageBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xMessageBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xMessageBufferSendFromISR() will set *pxHigherPriority-TaskWoken to pdTRUE. If xMessageBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *px-HigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Returns The number of bytes actually written to the message buffer. If the message buffer didn't have enough free space for the message to be stored then 0 is returned, otherwise xDataLengthBytes is returned.

xMessageBufferReceive (xMessageBuffer, pvRxData, xBufferLengthBytes, xTicksToWait)

Receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Example use:

```
void vAFunction( MessageBuffer_t xMessageBuffer )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

 // Receive the next message from the message buffer.  Wait in the Blocked
 // state (so not using any CPU processing time) for a maximum of 100ms for
 // a message to become available.
 xReceivedBytes = xMessageBufferReceive( xMessageBuffer,
                                         ( void * ) ucRxData,
                                         sizeof( ucRxData ),
                                         xBlockTime );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains a message that is xReceivedBytes long.  Process
     // the message here....
 }
}
```

Parameters

- **xMessageBuffer** –The handle of the message buffer from which a message is being received.
- **pvRxData** –A pointer to the buffer into which the received message is to be copied.
- **xBufferLengthBytes** –The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
- **xTicksToWait** –The maximum amount of time the task should remain in the Blocked state to wait for a message, should the message buffer be empty. xMessageBufferReceive() will return immediately if xTicksToWait is zero and the message buffer is empty. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided IN-

CLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Returns The length, in bytes, of the message read from the message buffer, if any. If xMessageBufferReceive() times out before a message became available then zero is returned. If the length of the message is greater than xBufferLengthBytes then the message will be left in the message buffer and zero is returned.

xMessageBufferReceiveFromISR(xMessageBuffer, pvRxData, xBufferLengthBytes, pxHigherPriorityTaskWoken)

An interrupt safe version of the API function that receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Example use:

```
// A message buffer that has already been created.
MessageBuffer_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;  // Initialised to pdFALSE.

 // Receive the next message from the message buffer.
 xReceivedBytes = xMessageBufferReceiveFromISR( xMessageBuffer,
                                                ( void * ) ucRxData,
                                                sizeof( ucRxData ),
                                                &xHigherPriorityTaskWoken );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains a message that is xReceivedBytes long.  Process
     // the message here....
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xMessageBufferReceiveFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Parameters

- **xMessageBuffer** –The handle of the message buffer from which a message is being received.
- **pvRxData** –A pointer to the buffer into which the received message is to be copied.
- **xBufferLengthBytes** –The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
- **pxHigherPriorityTaskWoken** –It is possible that a message buffer will have a task blocked on it waiting for space to become available. Calling xMessageBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xMessageBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xMessageBufferReceiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xMessageBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Returns The length, in bytes, of the message read from the message buffer, if any.

vMessageBufferDelete(xMessageBuffer)

Deletes a message buffer that was previously created using a call to xMessageBufferCreate() or xMessageBufferCreateStatic(). If the message buffer was created using dynamic memory (that is, by xMessageBufferCreate()), then the allocated memory is freed.

A message buffer handle must not be used after the message buffer has been deleted.

Parameters

- **xMessageBuffer** –The handle of the message buffer to be deleted.

xMessageBufferIsFull(xMessageBuffer)

Tests to see if a message buffer is full. A message buffer is full if it cannot accept any more messages, of any size, until space is made available by a message being removed from the message buffer.

Parameters

- **xMessageBuffer** –The handle of the message buffer being queried.

Returns If the message buffer referenced by xMessageBuffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

xMessageBufferIsEmpty(xMessageBuffer)

Tests to see if a message buffer is empty (does not contain any messages).

Parameters

- **xMessageBuffer** –The handle of the message buffer being queried.

Returns If the message buffer referenced by xMessageBuffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

xMessageBufferReset(xMessageBuffer)

Resets a message buffer to its initial empty state, discarding any message it contained.

A message buffer can only be reset if there are no tasks blocked on it.

Parameters

- **xMessageBuffer** –The handle of the message buffer being reset.

Returns If the message buffer was reset then pdPASS is returned. If the message buffer could not be reset because either there was a task blocked on the message queue to wait for space to become available, or to wait for a a message to be available, then pdFAIL is returned.

xMessageBufferSpaceAvailable(xMessageBuffer)

Returns the number of bytes of free space in the message buffer.

Parameters

- **xMessageBuffer** –The handle of the message buffer being queried.

Returns The number of bytes that can be written to the message buffer before the message buffer would be full. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message' s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so if xMessageBufferSpacesAvailable() returns 10, then the size of the largest message that can be written to the message buffer is 6 bytes.

xMessageBufferSpacesAvailable(xMessageBuffer)

xMessageBufferNextLengthBytes(xMessageBuffer)

Returns the length (in bytes) of the next message in a message buffer. Useful if xMessageBufferReceive() returned 0 because the size of the buffer passed into xMessageBufferReceive() was too small to hold the next message.

Parameters

- **xMessageBuffer** –The handle of the message buffer being queried.

Returns The length (in bytes) of the next message in the message buffer, or 0 if the message buffer is empty.

xMessageBufferSendCompletedFromISR(xMessageBuffer, pxHigherPriorityTaskWoken)

For advanced users only.

The sbSEND_COMPLETED() macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbSEND_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xMessageBufferSendCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbSEND_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

Parameters

- **xMessageBuffer** –The handle of the stream buffer to which data was written.
- **pxHigherPriorityTaskWoken** –*pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xMessageBufferSendCompletedFromISR(). If calling xMessageBufferSendCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

Returns If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

xMessageBufferReceiveCompletedFromISR(xMessageBuffer, pxHigherPriorityTaskWoken)

For advanced users only.

The sbRECEIVE_COMPLETED() macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbRECEIVE_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xMessageBufferReceiveCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbRECEIVE_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

Parameters

- **xMessageBuffer** –The handle of the stream buffer from which data was read.
- **pxHigherPriorityTaskWoken** –*pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xMessageBufferReceiveCompletedFromISR(). If calling xMessageBufferReceiveCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

Returns If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

Type Definitions

typedef void ***MessageBufferHandle_t**

Type by which message buffers are referenced. For example, a call to xMessageBufferCreate() returns an MessageBufferHandle_t variable that can then be used as a parameter to xMessageBufferSend(), xMessageBufferReceive(), etc.

2.10.12 FreeRTOS (Supplemental Features)

ESP-IDF provides multiple features to supplement the features offered by FreeRTOS. These supplemental features are available on all FreeRTOS implementations supported by ESP-IDF (i.e., ESP-IDF FreeRTOS and Amazon SMP FreeRTOS). This document describes these supplemental features and is split into the following sections:

Contents

- *FreeRTOS (Supplemental Features)*
  - *Overview*
  - *Ring Buffers*
  - *ESP-IDF Tick and Idle Hooks*
  - *TLSP Deletion Callbacks*
  - *Component Specific Properties*
  - *API Reference*

Overview

ESP-IDF adds various new features to supplement the capabilities of FreeRTOS as follows:

- **Ring buffers**: Ring buffers provide a FIFO buffer that can accept entries of arbitrary lengths.
- **ESP-IDF Tick and Idle Hooks**: ESP-IDF provides multiple custom tick interrupt hooks and idle task hooks that are more numerous and more flexible when compared to FreeRTOS tick and idle hooks.
- **Thread Local Storage Pointer (TLSP) Deletion Callbacks**: TLSP Deletion callbacks are run automatically when a task is deleted, thus allowing users to clean up their TLSPs automatically.
- **Component Specific Properties**: Currently added only one component specific property `ORIG_INCLUDE_PATH`.

Ring Buffers

FreeRTOS provides stream buffers and message buffers as the primary mechanisms to send arbitrarily sized data between tasks and ISRs. However, FreeRTOS stream buffers and message buffers have the following limitations:

- Strictly single sender and single receiver
- Data is passed by copy
- Unable to reserve buffer space for a deferred send (i.e., send acquire)

Therefore, ESP-IDF provides a separate ring buffer implementation to address the issues above. ESP-IDF ring buffers are strictly FIFO buffers that supports arbitrarily sized items. Ring buffers are a more memory efficient alternative to FreeRTOS queues in situations where the size of items is variable. The capacity of a ring buffer is not measured by the number of items it can store, but rather by the amount of memory used for storing items. The ring buffer provides APIs to send an item, or to allocate space for an item in the ring buffer to be filled manually by the user. For efficiency reasons, **items are always retrieved from the ring buffer by reference**. As a result, all retrieved

items *must also be returned* to the ring buffer by using `vRingbufferReturnItem()` or `vRingbufferReturnItemFromISR()`, in order for them to be removed from the ring buffer completely. The ring buffers are split into the three following types:

No-Split buffers will guarantee that an item is stored in contiguous memory and will not attempt to split an item under any circumstances. Use No-Split buffers when items must occupy contiguous memory. *Only this buffer type allows you to get the data item address and write to the item by yourself.* Refer the documentation of the functions `xRingbufferSendAcquire()` and `xRingbufferSendComplete()` for more details.

Allow-Split buffers will allow an item to be split in two parts when wrapping around the end of the buffer if there is enough space at the tail and the head of the buffer combined to store the item. Allow-Split buffers are more memory efficient than No-Split buffers but can return an item in two parts when retrieving.

Byte buffers do not store data as separate items. All data is stored as a sequence of bytes, and any number of bytes can be sent or retrieved each time. Use byte buffers when separate items do not need to be maintained (e.g. a byte stream).

Note: No-Split buffers and Allow-Split buffers will always store items at 32-bit aligned addresses. Therefore, when retrieving an item, the item pointer is guaranteed to be 32-bit aligned. This is useful especially when you need to send some data to the DMA.

Note: Each item stored in No-Split or Allow-Split buffers will **require an additional 8 bytes for a header**. Item sizes will also be rounded up to a 32-bit aligned size (multiple of 4 bytes), however the true item size is recorded within the header. The sizes of No-Split and Allow-Split buffers will also be rounded up when created.

Usage The following example demonstrates the usage of `xRingbufferCreate()` and `xRingbufferSend()` to create a ring buffer and then send an item to it.

```
#include "freertos/ringbuf.h"
static char tx_item[] = "test_item";

...

    //Create ring buffer
    RingbufHandle_t buf_handle;
    buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
    if (buf_handle == NULL) {
        printf("Failed to create ring buffer\n");
    }

    //Send an item
    UBaseType_t res =  xRingbufferSend(buf_handle, tx_item, sizeof(tx_item), pdMS_
↪TO_TICKS(1000));
    if (res != pdTRUE) {
        printf("Failed to send item\n");
    }
```

The following example demonstrates the usage of `xRingbufferSendAcquire()` and `xRingbufferSendComplete()` instead of `xRingbufferSend()` to acquire memory on the ring buffer (of type *RINGBUF_TYPE_NOSPLIT*) and then send an item to it. This adds one more step, but allows getting the address of the memory to write to, and writing to the memory yourself.

```
#include "freertos/ringbuf.h"
#include "soc/lldesc.h"

typedef struct {
    lldesc_t dma_desc;
```

(continues on next page)

(continued from previous page)

```
    uint8_t buf[1];
} dma_item_t;

#define DMA_ITEM_SIZE(N) (sizeof(lldesc_t)+(((N)+3)&(~3)))

...

    //Retrieve space for DMA descriptor and corresponding data buffer
    //This has to be done with SendAcquire, or the address may be different when
↪we copy
    dma_item_t item;
    UBaseType_t res =  xRingbufferSendAcquire(buf_handle,
                       &item, DMA_ITEM_SIZE(buffer_size), pdMS_TO_TICKS(1000));
    if (res != pdTRUE) {
        printf("Failed to acquire memory for item\n");
    }
    item->dma_desc = (lldesc_t) {
        .size = buffer_size,
        .length = buffer_size,
        .eof = 0,
        .owner = 1,
        .buf = &item->buf,
    };
    //Actually send to the ring buffer for consumer to use
    res = xRingbufferSendComplete(buf_handle, &item);
    if (res != pdTRUE) {
        printf("Failed to send item\n");
    }
```

The following example demonstrates retrieving and returning an item from a **No-Split ring buffer** using *xRingbufferReceive()* and *vRingbufferReturnItem()*

```
...

    //Receive an item from no-split ring buffer
    size_t item_size;
    char *item = (char *)xRingbufferReceive(buf_handle, &item_size, pdMS_TO_
↪TICKS(1000));

    //Check received item
    if (item != NULL) {
        //Print item
        for (int i = 0; i < item_size; i++) {
            printf("%c", item[i]);
        }
        printf("\n");
        //Return Item
        vRingbufferReturnItem(buf_handle, (void *)item);
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

The following example demonstrates retrieving and returning an item from an **Allow-Split ring buffer** using *xRingbufferReceiveSplit()* and *vRingbufferReturnItem()*

```
...

    //Receive an item from allow-split ring buffer
    size_t item_size1, item_size2;
    char *item1, *item2;
```

(continues on next page)

(continued from previous page)

```
    BaseType_t ret = xRingbufferReceiveSplit(buf_handle, (void **)&item1, (void
↪**)&item2, &item_size1, &item_size2, pdMS_TO_TICKS(1000));

    //Check received item
    if (ret == pdTRUE && item1 != NULL) {
        for (int i = 0; i < item_size1; i++) {
            printf("%c", item1[i]);
        }
        vRingbufferReturnItem(buf_handle, (void *)item1);
        //Check if item was split
        if (item2 != NULL) {
            for (int i = 0; i < item_size2; i++) {
                printf("%c", item2[i]);
            }
            vRingbufferReturnItem(buf_handle, (void *)item2);
        }
        printf("\n");
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

The following example demonstrates retrieving and returning an item from a **byte buffer** using *xRingbufferReceiveUpTo()* and *vRingbufferReturnItem()*

```
...

    //Receive data from byte buffer
    size_t item_size;
    char *item = (char *)xRingbufferReceiveUpTo(buf_handle, &item_size, pdMS_TO_
↪TICKS(1000), sizeof(tx_item));

    //Check received data
    if (item != NULL) {
        //Print item
        for (int i = 0; i < item_size; i++) {
            printf("%c", item[i]);
        }
        printf("\n");
        //Return Item
        vRingbufferReturnItem(buf_handle, (void *)item);
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

For ISR safe versions of the functions used above, call *xRingbufferSendFromISR()*, *xRingbufferReceiveFromISR()*, *xRingbufferReceiveSplitFromISR()*, *xRingbufferReceiveUpToFromISR()*, and *vRingbufferReturnItemFromISR()*

Note: Two calls to RingbufferReceive[UpTo][FromISR]() are required if the bytes wraps around the end of the ring buffer.

Sending to Ring Buffer The following diagrams illustrate the differences between No-Split and Allow-Split buffers as compared to byte buffers with regard to sending items/data. The diagrams assume that three items of sizes **18, 3, and 27 bytes** are sent respectively to a **buffer of 128 bytes**.

For No-Split and Allow-Split buffers, a header of 8 bytes precedes every data item. Furthermore, the space occupied by each item is **rounded up to the nearest 32-bit aligned size** in order to maintain overall 32-bit alignment.

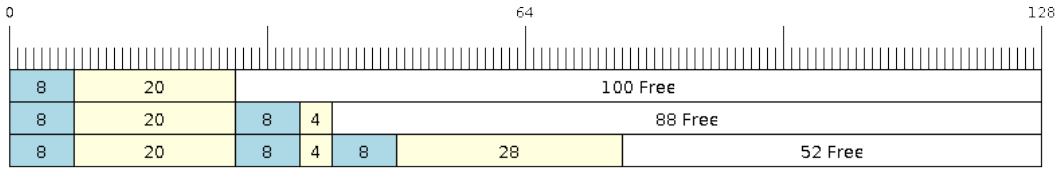


Fig. 32: Sending items to No-Split or Allow-Split ring buffers

However, the true size of the item is recorded inside the header which will be returned when the item is retrieved.

Referring to the diagram above, the 18, 3, and 27 byte items are **rounded up to 20, 4, and 28 bytes** respectively. An 8 byte header is then added in front of each item.

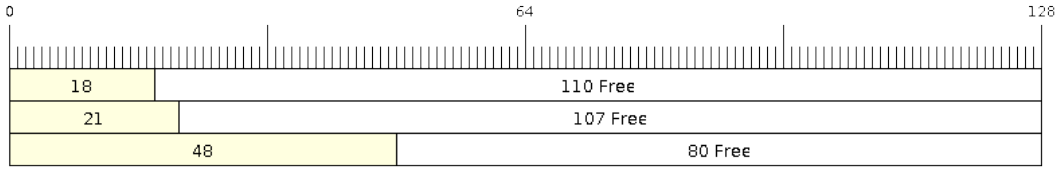


Fig. 33: Sending items to byte buffers

Byte buffers treat data as a sequence of bytes and does not incur any overhead (no headers). As a result, all data sent to a byte buffer is merged into a single item.

Referring to the diagram above, the 18, 3, and 27 byte items are sequentially written to the byte buffer and **merged into a single item of 48 bytes**.

Using SendAcquire and SendComplete Items in No-Split buffers are acquired (by `SendAcquire`) in strict FIFO order and must be sent to the buffer by `SendComplete` for the data to be accessible by the consumer. Multiple items can be sent or acquired without calling `SendComplete`, and the items do not necessarily need to be completed in the order they were acquired. However, the receiving of data items must occur in FIFO order, therefore not calling `SendComplete` for the earliest acquired item will prevent the subsequent items from being received.

The following diagrams illustrate what will happen when `SendAcquire` and `SendComplete` don't happen in the same order. At the beginning, there is already a data item of 16 bytes sent to the ring buffer. Then `SendAcquire` is called to acquire space of 20, 8, 24 bytes on the ring buffer.

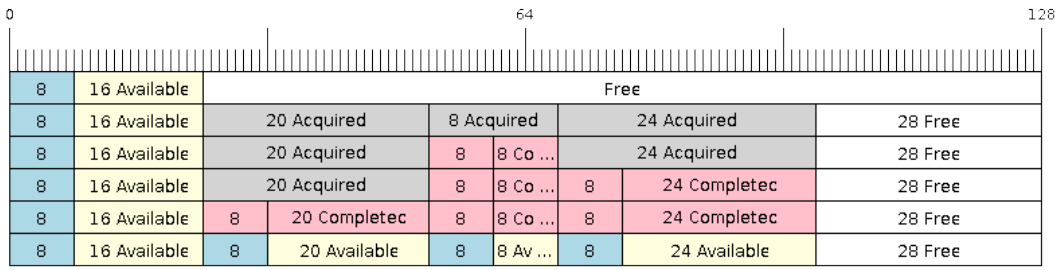


Fig. 34: SendAcquire/SendComplete items in No-Split ring buffers

After that, we fill (use) the buffers, and send them to the ring buffer by `SendComplete` in the order of 8, 24, 20. When 8 bytes and 24 bytes data are sent, the consumer still can only get the 16 bytes data item. Hence, if

`SendComplete` is not called for the 20 bytes, it will not be available, nor will the data items following the 20 bytes item.

When the 20 bytes item is finally completed, all the 3 data items can be received now, in the order of 20, 8, 24 bytes, right after the 16 bytes item existing in the buffer at the beginning.

Allow-Split buffers and byte buffers do not allow using `SendAcquire` or `SendComplete` since acquired buffers are required to be complete (not wrapped).

Wrap around The following diagrams illustrate the differences between No-Split, Allow-Split, and byte buffers when a sent item requires a wrap around. The diagrams assume a buffer of **128 bytes** with **56 bytes of free space that wraps around** and a sent item of **28 bytes**.

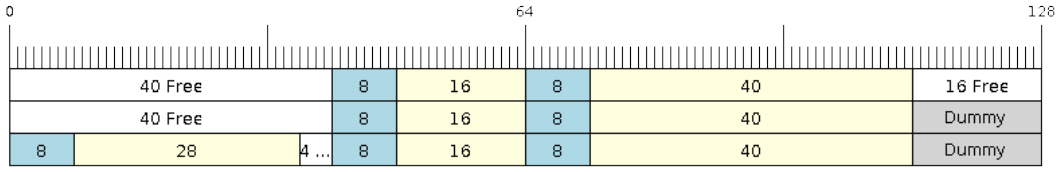


Fig. 35: Wrap around in No-Split buffers

No-Split buffers will **only store an item in continuous free space and will not split an item under any circumstances**. When the free space at the tail of the buffer is insufficient to completely store the item and its header, the free space at the tail will be **marked as dummy data**. The buffer will then wrap around and store the item in the free space at the head of the buffer.

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore, the 16 bytes is marked as dummy data and the item is written to the free space at the head of the buffer instead.

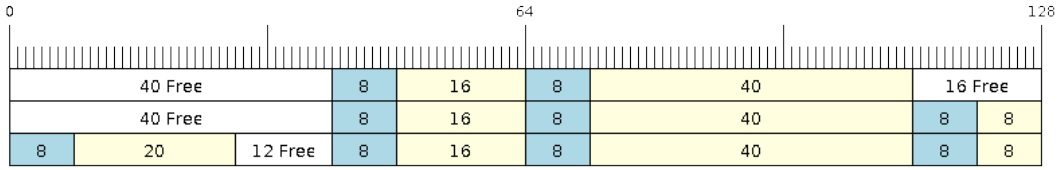


Fig. 36: Wrap around in Allow-Split buffers

Allow-Split buffers will attempt to **split the item into two parts** when the free space at the tail of the buffer is insufficient to store the item data and its header. Both parts of the split item will have their own headers (therefore incurring an extra 8 bytes of overhead).

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore, the item is split into two parts (8 and 20 bytes) and written as two parts to the buffer.

Note: Allow-Split buffers treat both parts of the split item as two separate items, therefore call *`xRingbufferReceiveSplit()`* instead of *`xRingbufferReceive()`* to receive both parts of a split item in a thread safe manner.

Byte buffers will **store as much data as possible into the free space at the tail of buffer**. The remaining data will then be stored in the free space at the head of the buffer. No overhead is incurred when wrapping around in byte buffers.

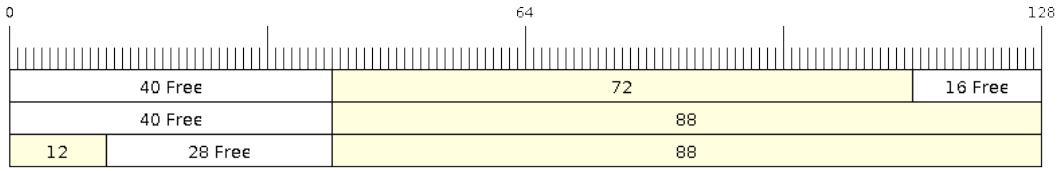


Fig. 37: Wrap around in byte buffers

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to completely store the 28 bytes of data. Therefore, the 16 bytes of free space is filled with data, and the remaining 12 bytes are written to the free space at the head of the buffer. The buffer now contains data in two separate continuous parts, and each continuous part will be treated as a separate item by the byte buffer.

Retrieving/Returning The following diagrams illustrate the differences between No-Split and Allow-Split buffers as compared to byte buffers in retrieving and returning data.

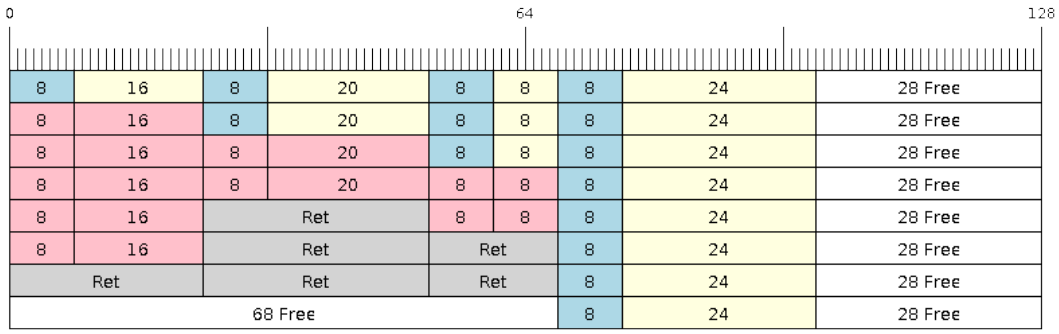


Fig. 38: Retrieving/Returning items in No-Split and Allow-Split ring buffers

Items in No-Split buffers and Allow-Split buffers are **retrieved in strict FIFO order** and **must be returned** for the occupied space to be freed. Multiple items can be retrieved before returning, and the items do not necessarily need to be returned in the order they were retrieved. However, the freeing of space must occur in FIFO order, therefore not returning the earliest retrieved item will prevent the space of subsequent items from being freed.

Referring to the diagram above, the **16, 20, and 8 byte items are retrieved in FIFO order**. However, the items are not returned in the order they were retrieved. First, the 20 byte item is returned followed by the 8 byte and the 16 byte items. The space is not freed until the first item, i.e., the 16 byte item is returned.

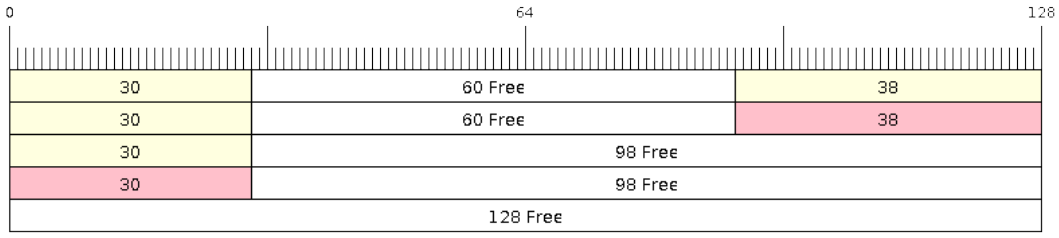


Fig. 39: Retrieving/Returning data in byte buffers

Byte buffers **do not allow multiple retrievals before returning** (every retrieval must be followed by a return before another retrieval is permitted). When using *xRingbufferReceive()* or *xRingbufferReceiveFromISR()*, all continuous stored data will be retrieved. *xRingbufferReceiveUpTo()* or *xRingbufferReceiveUpToFromISR()* can be used to restrict the maximum number of bytes retrieved. Since every retrieval must be followed by a return, the space will be freed as soon as the data is returned.

Referring to the diagram above, the 38 bytes of continuous stored data at the tail of the buffer is retrieved, returned, and freed. The next call to *xRingbufferReceive()* or *xRingbufferReceiveFromISR()* then wraps around and does the same to the 30 bytes of continuous stored data at the head of the buffer.

Ring Buffers with Queue Sets Ring buffers can be added to FreeRTOS queue sets using *xRingbufferAddToQueueSetRead()* such that every time a ring buffer receives an item or data, the queue set is notified. Once added to a queue set, every attempt to retrieve an item from a ring buffer should be preceded by a call to *xQueueSelectFromSet()*. To check whether the selected queue set member is the ring buffer, call *xRingbufferCanRead()*.

The following example demonstrates queue set usage with ring buffers.

```
#include "freertos/queue.h"
#include "freertos/ringbuf.h"

...

    //Create ring buffer and queue set
    RingbufHandle_t buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
    QueueSetHandle_t queue_set = xQueueCreateSet(3);

    //Add ring buffer to queue set
    if (xRingbufferAddToQueueSetRead(buf_handle, queue_set) != pdTRUE) {
        printf("Failed to add to queue set\n");
    }

...

    //Block on queue set
    QueueSetMemberHandle_t member = xQueueSelectFromSet(queue_set, pdMS_TO_
↪TICKS(1000));

    //Check if member is ring buffer
    if (member != NULL && xRingbufferCanRead(buf_handle, member) == pdTRUE) {
        //Member is ring buffer, receive item from ring buffer
        size_t item_size;
        char *item = (char *)xRingbufferReceive(buf_handle, &item_size, 0);

        //Handle item
        ...

    } else {
        ...
    }
```

Ring Buffers with Static Allocation The *xRingbufferCreateStatic()* can be used to create ring buffers with specific memory requirements (such as a ring buffer being allocated in external RAM). All blocks of memory used by a ring buffer must be manually allocated beforehand then passed to the *xRingbufferCreateStatic()* to be initialized as a ring buffer. These blocks include the following:

- The ring buffer's data structure of type *StaticRingbuffer_t*
- The ring buffer's storage area of size `xBufferSize`. Note that `xBufferSize` must be 32-bit aligned for No-Split and Allow-Split buffers.

The manner in which these blocks are allocated will depend on the users requirements (e.g. all blocks being statically declared, or dynamically allocated with specific capabilities such as external RAM).

Note: When deleting a ring buffer created via *xRingbufferCreateStatic()*, the function *vRingbufferDelete()* will not free any of the memory blocks. This must be done manually by the user after *vRingbufferDelete()* is called.

The code snippet below demonstrates a ring buffer being allocated entirely in external RAM.

```
#include "freertos/ringbuf.h"
#include "freertos/semphr.h"
#include "esp_heap_caps.h"

#define BUFFER_SIZE     400      //32-bit aligned size
#define BUFFER_TYPE     RINGBUF_TYPE_NOSPLIT
...

//Allocate ring buffer data structure and storage area into external RAM
StaticRingbuffer_t *buffer_struct = (StaticRingbuffer_t *)heap_caps_
↪malloc(sizeof(StaticRingbuffer_t), MALLOC_CAP_SPIRAM);
uint8_t *buffer_storage = (uint8_t *)heap_caps_malloc(sizeof(uint8_t)*BUFFER_SIZE,␣
↪MALLOC_CAP_SPIRAM);

//Create a ring buffer with manually allocated memory
RingbufHandle_t handle = xRingbufferCreateStatic(BUFFER_SIZE, BUFFER_TYPE, buffer_
↪storage, buffer_struct);

...

//Delete the ring buffer after used
vRingbufferDelete(handle);

//Manually free all blocks of memory
free(buffer_struct);
free(buffer_storage);
```

Priority Inversion Ideally, ring buffers can be used with multiple tasks in an SMP fashion where the **highest priority task will always be serviced first.** However due to the usage of binary semaphores in the ring buffer's underlying implementation, priority inversion may occur under very specific circumstances.

The ring buffer governs sending by a binary semaphore which is given whenever space is freed on the ring buffer. The highest priority task waiting to send will repeatedly take the semaphore until sufficient free space becomes available or until it times out. Ideally this should prevent any lower priority tasks from being serviced as the semaphore should always be given to the highest priority task.

However, in between iterations of acquiring the semaphore, there is a **gap in the critical section** which may permit another task (on the other core or with an even higher priority) to free some space on the ring buffer and as a result give the semaphore. Therefore, the semaphore will be given before the highest priority task can re-acquire the semaphore. This will result in the **semaphore being acquired by the second-highest priority task** waiting to send, hence causing priority inversion.

This side effect will not affect ring buffer performance drastically given if the number of tasks using the ring buffer simultaneously is low, and the ring buffer is not operating near maximum capacity.

ESP-IDF Tick and Idle Hooks

FreeRTOS allows applications to provide a tick hook and an idle hook at compile time:

- FreeRTOS tick hook can be enabled via the *CONFIG_FREERTOS_USE_TICK_HOOK* option. The application must provide the `void vApplicationTickHook( void )` callback.

- FreeRTOS idle hook can be enabled via the *CONFIG_FREERTOS_USE_IDLE_HOOK* option. The application must provide the `void vApplicationIdleHook( void )` callback.

However, the FreeRTOS tick hook and idle hook have the following draw backs:

- The FreeRTOS hooks are registered at compile time
- Only one of each hook can be registered
- On multi-core targets, the FreeRTOS hooks are symmetric, meaning each CPU's tick interrupt and idle tasks ends up calling the same hook.

Therefore, ESP-IDF tick and idle hooks are provided to supplement the features of FreeRTOS tick and idle hooks. The ESP-IDF hooks have the following features:

- The hooks can be registered and deregistered at run-time
- Multiple hooks can be registered (with a maximum of 8 hooks of each type per CPU)
- On multi-core targets, the hooks can be asymmetric, meaning different hooks can be registered to each CPU

ESP-IDF hooks can be registered and deregistered using the following API:

- For tick hooks:
  - Register using `esp_register_freertos_tick_hook()` or `esp_register_freertos_tick_hook_for_cpu()`
  - Deregister using `esp_deregister_freertos_tick_hook()` or `esp_deregister_freertos_tick_hook_for_cpu()`
- For idle hooks:
  - Register using `esp_register_freertos_idle_hook()` or `esp_register_freertos_idle_hook_for_cpu()`
  - Deregister using `esp_deregister_freertos_idle_hook()` or `esp_deregister_freertos_idle_hook_for_cpu()`

Note: The tick interrupt stays active while the cache is disabled, therefore any tick hook (FreeRTOS or ESP-IDF) functions must be placed in internal RAM. Please refer to the *SPI flash API documentation* for more details.

TLSP Deletion Callbacks

Vanilla FreeRTOS provides a Thread Local Storage Pointers (TLSP) feature. These are pointers stored directly in the Task Control Block (TCB) of a particular task. TLSPs allow each task to have its own unique set of pointers to data structures. Vanilla FreeRTOS expects users to…

- set a task's TLSPs by calling `vTaskSetThreadLocalStoragePointer()` after the task has been created.
- get a task's TLSPs by calling `pvTaskGetThreadLocalStoragePointer()` during the task's lifetime.
- free the memory pointed to by the TLSPs before the task is deleted.

However, there can be instances where users may want the freeing of TLSP memory to be automatic. Therefore, ESP-IDF provides the additional feature of TLSP deletion callbacks. These user provided deletion callbacks are called automatically when a task is deleted, thus allowing the TLSP memory to be cleaned up without needing to add the cleanup logic explicitly to the code of every task.

The TLSP deletion callbacks are set in a similar fashion to the TLSPs themselves.

- `vTaskSetThreadLocalStoragePointerAndDelCallback()` sets both a particular TLSP and its associated callback.
- Calling the Vanilla FreeRTOS function `vTaskSetThreadLocalStoragePointer()` will simply set the TLSP's associated Deletion Callback to *NULL* meaning that no callback will be called for that TLSP during task deletion.

When implementing TLSP callbacks, users should note the following:

- The callback **must never attempt to block or yield** and critical sections should be kept as short as possible

- The callback is called shortly before a deleted task's memory is freed. Thus, the callback can either be called from `vTaskDelete()` itself, or from the idle task.

Component Specific Properties

Besides standard component variables that are available with basic cmake build properties, FreeRTOS component also provides arguments (only one so far) for simpler integration with other modules:

- *ORIG_INCLUDE_PATH* - contains an absolute path to freertos root include folder. Thus instead of *#include "freertos/FreeRTOS.h"* you can refer to headers directly: *#include "FreeRTOS.h"*.

API Reference

Ring Buffer API

Header File

- components/esp_ringbuf/include/freertos/ringbuf.h

Functions

RingbufHandle_t **xRingbufferCreate**(size_t xBufferSize, *RingbufferType_t* xBufferType)

Create a ring buffer.

Note: xBufferSize of no-split/allow-split buffers will be rounded up to the nearest 32-bit aligned size.

Parameters

- **xBufferSize** –**[in]** Size of the buffer in bytes. Note that items require space for a header in no-split/allow-split buffers
- **xBufferType** –**[in]** Type of ring buffer, see documentation.

Returns A handle to the created ring buffer, or NULL in case of error.

RingbufHandle_t **xRingbufferCreateNoSplit**(size_t xItemSize, size_t xItemNum)

Create a ring buffer of type RINGBUF_TYPE_NOSPLIT for a fixed item_size.

This API is similar to xRingbufferCreate(), but it will internally allocate additional space for the headers.

Parameters

- **xItemSize** –**[in]** Size of each item to be put into the ring buffer
- **xItemNum** –**[in]** Maximum number of items the buffer needs to hold simultaneously

Returns A RingbufHandle_t handle to the created ring buffer, or NULL in case of error.

RingbufHandle_t **xRingbufferCreateStatic**(size_t xBufferSize, *RingbufferType_t* xBufferType, uint8_t *pucRingbufferStorage, *StaticRingbuffer_t* *pxStaticRingbuffer)

Create a ring buffer but manually provide the required memory.

Note: xBufferSize of no-split/allow-split buffers MUST be 32-bit aligned.

Parameters

- **xBufferSize** –**[in]** Size of the buffer in bytes.
- **xBufferType** –**[in]** Type of ring buffer, see documentation
- **pucRingbufferStorage** –**[in]** Pointer to the ring buffer's storage area. Storage area must have the same size as specified by xBufferSize

- **pxStaticRingbuffer** –**[in]** Pointed to a struct of type StaticRingbuffer_t which will be used to hold the ring buffer' s data structure

Returns A handle to the created ring buffer

BaseType_t **xRingbufferSend**(*RingbufHandle_t* xRingbuffer, const void *pvItem, size_t xItemSize, TickType_t xTicksToWait)

Insert an item into the ring buffer.

Attempt to insert an item into the ring buffer. This function will block until enough free space is available or until it times out.

Note: For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Parameters

- **xRingbuffer** –**[in]** Ring buffer to insert the item into
- **pvItem** –**[in]** Pointer to data to insert. NULL is allowed if xItemSize is 0.
- **xItemSize** –**[in]** Size of data to insert.
- **xTicksToWait** –**[in]** Ticks to wait for room in the ring buffer.

Returns

- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the buffer

BaseType_t **xRingbufferSendFromISR**(*RingbufHandle_t* xRingbuffer, const void *pvItem, size_t xItemSize, BaseType_t *pxHigherPriorityTaskWoken)

Insert an item into the ring buffer in an ISR.

Attempt to insert an item into the ring buffer from an ISR. This function will return immediately if there is insufficient free space in the buffer.

Note: For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Parameters

- **xRingbuffer** –**[in]** Ring buffer to insert the item into
- **pvItem** –**[in]** Pointer to data to insert. NULL is allowed if xItemSize is 0.
- **xItemSize** –**[in]** Size of data to insert.
- **pxHigherPriorityTaskWoken** –**[out]** Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

Returns

- pdTRUE if succeeded
- pdFALSE when the ring buffer does not have space.

BaseType_t **xRingbufferSendAcquire**(*RingbufHandle_t* xRingbuffer, void **ppvItem, size_t xItemSize, TickType_t xTicksToWait)

Acquire memory from the ring buffer to be written to by an external source and to be sent later.

Attempt to allocate buffer for an item to be sent into the ring buffer. This function will block until enough free space is available or until it times out.

The item, as well as the following items `SendAcquire` or `Send` after it, will not be able to be read from the ring buffer until this item is actually sent into the ring buffer.

Note: Only applicable for no-split ring buffers now, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit

aligned fashion.

Parameters
- **xRingbuffer** –**[in]** Ring buffer to allocate the memory
- **ppvItem** –**[out]** Double pointer to memory acquired (set to NULL if no memory were retrieved)
- **xItemSize** –**[in]** Size of item to acquire.
- **xTicksToWait** –**[in]** Ticks to wait for room in the ring buffer.

Returns
- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the buffer

BaseType_t **xRingbufferSendComplete**(*RingbufHandle_t* xRingbuffer, void *pvItem)

Actually send an item into the ring buffer allocated before by `xRingbufferSendAcquire`.

Note: Only applicable for no-split ring buffers. Only call for items allocated by `xRingbufferSendAcquire`.

Parameters
- **xRingbuffer** –**[in]** Ring buffer to insert the item into
- **pvItem** –**[in]** Pointer to item in allocated memory to insert.

Returns
- pdTRUE if succeeded
- pdFALSE if fail for some reason.

void *__xRingbufferReceive__(*RingbufHandle_t* xRingbuffer, size_t *pxItemSize, TickType_t xTicksToWait)

Retrieve an item from the ring buffer.

Attempt to retrieve an item from the ring buffer. This function will block until an item is available or until it times out.

Note: A call to vRingbufferReturnItem() is required after this to free the item retrieved.

Parameters
- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **pxItemSize** –**[out]** Pointer to a variable to which the size of the retrieved item will be written.
- **xTicksToWait** –**[in]** Ticks to wait for items in the ring buffer.

Returns
- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL on timeout, *pxItemSize is untouched in that case.

void *__xRingbufferReceiveFromISR__(*RingbufHandle_t* xRingbuffer, size_t *pxItemSize)

Retrieve an item from the ring buffer in an ISR.

Attempt to retrieve an item from the ring buffer. This function returns immediately if there are no items available for retrieval

Note: A call to vRingbufferReturnItemFromISR() is required after this to free the item retrieved.

Note: Byte buffers do not allow multiple retrievals before returning an item

Note: Two calls to RingbufferReceiveFromISR() are required if the bytes wrap around the end of the ring buffer.

Parameters

- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **pxItemSize** –**[out]** Pointer to a variable to which the size of the retrieved item will be written.

Returns

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL when the ring buffer is empty, *pxItemSize is untouched in that case.

BaseType_t **xRingbufferReceiveSplit**(*RingbufHandle_t* xRingbuffer, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize, TickType_t xTicksToWait)

Retrieve a split item from an allow-split ring buffer.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retried. If the item is split, both parts will be retrieved. This function will block until an item is available or until it times out.

Note: Call(s) to vRingbufferReturnItem() is required after this to free up the item(s) retrieved.

Note: This function should only be called on allow-split buffers

Parameters

- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **ppvHeadItem** –**[out]** Double pointer to first part (set to NULL if no items were retrieved)
- **ppvTailItem** –**[out]** Double pointer to second part (set to NULL if item is not split)
- **pxHeadItemSize** –**[out]** Pointer to size of first part (unmodified if no items were retrieved)
- **pxTailItemSize** –**[out]** Pointer to size of second part (unmodified if item is not split)
- **xTicksToWait** –**[in]** Ticks to wait for items in the ring buffer.

Returns

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

BaseType_t **xRingbufferReceiveSplitFromISR**(*RingbufHandle_t* xRingbuffer, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize)

Retrieve a split item from an allow-split ring buffer in an ISR.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retried. If the item is split, both parts will be retrieved. This function returns immediately if there are no items available for retrieval

Note: Calls to vRingbufferReturnItemFromISR() is required after this to free up the item(s) retrieved.

Note: This function should only be called on allow-split buffers

Parameters

- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **ppvHeadItem** –**[out]** Double pointer to first part (set to NULL if no items were retrieved)
- **ppvTailItem** –**[out]** Double pointer to second part (set to NULL if item is not split)
- **pxHeadItemSize** –**[out]** Pointer to size of first part (unmodified if no items were retrieved)
- **pxTailItemSize** –**[out]** Pointer to size of second part (unmodified if item is not split)

Returns

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

void ***xRingbufferReceiveUpTo**(*RingbufHandle_t* xRingbuffer, size_t *pxItemSize, TickType_t xTicksToWait, size_t xMaxSize)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve.

Attempt to retrieve data from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will block until there is data available for retrieval or until it times out.

Note: A call to vRingbufferReturnItem() is required after this to free up the data retrieved.

Note: This function should only be called on byte buffers

Note: Byte buffers do not allow multiple retrievals before returning an item

Note: Two calls to RingbufferReceiveUpTo() are required if the bytes wrap around the end of the ring buffer.

Parameters

- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **pxItemSize** –**[out]** Pointer to a variable to which the size of the retrieved item will be written.
- **xTicksToWait** –**[in]** Ticks to wait for items in the ring buffer.
- **xMaxSize** –**[in]** Maximum number of bytes to return.

Returns

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL on timeout, *pxItemSize is untouched in that case.

void ***xRingbufferReceiveUpToFromISR**(*RingbufHandle_t* xRingbuffer, size_t *pxItemSize, size_t xMaxSize)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve. Call this from an ISR.

Attempt to retrieve bytes from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will return immediately if there is no data available for retrieval.

Note: A call to vRingbufferReturnItemFromISR() is required after this to free up the data received.

Note: This function should only be called on byte buffers

Note: Byte buffers do not allow multiple retrievals before returning an item

Parameters

- **xRingbuffer** –**[in]** Ring buffer to retrieve the item from
- **pxItemSize** –**[out]** Pointer to a variable to which the size of the retrieved item will be written.
- **xMaxSize** –**[in]** Maximum number of bytes to return.

Returns

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL when the ring buffer is empty, *pxItemSize is untouched in that case.

void **vRingbufferReturnItem**(*RingbufHandle_t* xRingbuffer, void *pvItem)

Return a previously-retrieved item to the ring buffer.

Note: If a split item is retrieved, both parts should be returned by calling this function twice

---

Parameters

- **xRingbuffer** –**[in]** Ring buffer the item was retrieved from
- **pvItem** –**[in]** Item that was received earlier

void **vRingbufferReturnItemFromISR**(*RingbufHandle_t* xRingbuffer, void *pvItem, BaseType_t *pxHigherPriorityTaskWoken)

Return a previously-retrieved item to the ring buffer from an ISR.

Note: If a split item is retrieved, both parts should be returned by calling this function twice

---

Parameters

- **xRingbuffer** –**[in]** Ring buffer the item was retrieved from
- **pvItem** –**[in]** Item that was received earlier
- **pxHigherPriorityTaskWoken** –**[out]** Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

void **vRingbufferDelete**(*RingbufHandle_t* xRingbuffer)

Delete a ring buffer.

Note: This function will not deallocate any memory if the ring buffer was created using xRingbufferCreateStatic(). Deallocation must be done manually be the user.

Parameters **xRingbuffer** –**[in]** Ring buffer to delete

size_t **xRingbufferGetMaxItemSize**(*RingbufHandle_t* xRingbuffer)

Get maximum size of an item that can be placed in the ring buffer.

This function returns the maximum size an item can have if it was placed in an empty ring buffer.

Note: The max item size for a no-split buffer is limited to ((buffer_size/2)-header_size). This limit is imposed so that an item of max item size can always be sent to an empty no-split buffer regardless of the internal positions of the buffer' s read/write/free pointers.

Parameters **xRingbuffer** –**[in]** Ring buffer to query
Returns Maximum size, in bytes, of an item that can be placed in a ring buffer.

size_t **xRingbufferGetCurFreeSize**(*RingbufHandle_t* xRingbuffer)

Get current free size available for an item/data in the buffer.

This gives the real time free space available for an item/data in the ring buffer. This represents the maximum size an item/data can have if it was currently sent to the ring buffer.

Note: An empty no-split buffer has a max current free size for an item that is limited to ((buffer_size/2)-header_size). See API reference for xRingbufferGetMaxItemSize().

Warning: This API is not thread safe. So, if multiple threads are accessing the same ring buffer, it is the application's responsibility to ensure atomic access to this API and the subsequent Send

Parameters **xRingbuffer** –**[in]** Ring buffer to query
Returns Current free size, in bytes, available for an entry

BaseType_t **xRingbufferAddToQueueSetRead**(*RingbufHandle_t* xRingbuffer, *QueueSetHandle_t* xQueueSet)

Add the ring buffer's read semaphore to a queue set.

The ring buffer's read semaphore indicates that data has been written to the ring buffer. This function adds the ring buffer's read semaphore to a queue set.

Parameters
- **xRingbuffer** –**[in]** Ring buffer to add to the queue set
- **xQueueSet** –**[in]** Queue set to add the ring buffer's read semaphore to

Returns
- pdTRUE on success, pdFALSE otherwise

BaseType_t **xRingbufferCanRead**(*RingbufHandle_t* xRingbuffer, *QueueSetMemberHandle_t* xMember)

Check if the selected queue set member is the ring buffer's read semaphore.

This API checks if queue set member returned from xQueueSelectFromSet() is the read semaphore of this ring buffer. If so, this indicates the ring buffer has items waiting to be retrieved.

Parameters
- **xRingbuffer** –**[in]** Ring buffer which should be checked
- **xMember** –**[in]** Member returned from xQueueSelectFromSet

Returns
- pdTRUE when semaphore belongs to ring buffer
- pdFALSE otherwise.

BaseType_t **xRingbufferRemoveFromQueueSetRead**(*RingbufHandle_t* xRingbuffer, *QueueSetHandle_t* xQueueSet)

Remove the ring buffer's read semaphore from a queue set.

This specifically removes a ring buffer's read semaphore from a queue set. The read semaphore is used to indicate when data has been written to the ring buffer

Parameters
- **xRingbuffer** –**[in]** Ring buffer to remove from the queue set
- **xQueueSet** –**[in]** Queue set to remove the ring buffer's read semaphore from

Returns
- pdTRUE on success
- pdFALSE otherwise

void **vRingbufferGetInfo**(*RingbufHandle_t* xRingbuffer, UBaseType_t *uxFree, UBaseType_t *uxRead, UBaseType_t *uxWrite, UBaseType_t *uxAcquire, UBaseType_t *uxItemsWaiting)

Get information about ring buffer status.

Get information of a ring buffer's current status such as free/read/write/acquire pointer positions, and number of items waiting to be retrieved. Arguments can be set to NULL if they are not required.

Parameters

- **xRingbuffer** –**[in]** Ring buffer to remove from the queue set
- **uxFree** –**[out]** Pointer use to store free pointer position
- **uxRead** –**[out]** Pointer use to store read pointer position
- **uxWrite** –**[out]** Pointer use to store write pointer position
- **uxAcquire** –**[out]** Pointer use to store acquire pointer position
- **uxItemsWaiting** –**[out]** Pointer use to store number of items (bytes for byte buffer) waiting to be retrieved

void **xRingbufferPrintInfo**(*RingbufHandle_t* xRingbuffer)

Debugging function to print the internal pointers in the ring buffer.

Parameters **xRingbuffer** –Ring buffer to show

Structures

struct **xSTATIC_RINGBUFFER**

Struct that is equivalent in size to the ring buffer's data structure.

The contents of this struct are not meant to be used directly. This structure is meant to be used when creating a statically allocated ring buffer where this struct is of the exact size required to store a ring buffer's control data structure.

Type Definitions

typedef void ***RingbufHandle_t**

Type by which ring buffers are referenced. For example, a call to xRingbufferCreate() returns a RingbufHandle_t variable that can then be used as a parameter to xRingbufferSend(), xRingbufferReceive(), etc.

typedef struct *xSTATIC_RINGBUFFER* **StaticRingbuffer_t**

Struct that is equivalent in size to the ring buffer's data structure.

The contents of this struct are not meant to be used directly. This structure is meant to be used when creating a statically allocated ring buffer where this struct is of the exact size required to store a ring buffer's control data structure.

Enumerations

enum **RingbufferType_t**

Values:

enumerator **RINGBUF_TYPE_NOSPLIT**

No-split buffers will only store an item in contiguous memory and will never split an item. Each item requires an 8 byte overhead for a header and will always internally occupy a 32-bit aligned size of space.

enumerator **RINGBUF_TYPE_ALLOWSPLIT**

Allow-split buffers will split an item into two parts if necessary in order to store it. Each item requires an 8 byte overhead for a header, splitting incurs an extra header. Each item will always internally occupy a 32-bit aligned size of space.

enumerator **RINGBUF_TYPE_BYTEBUF**

Byte buffers store data as a sequence of bytes and do not maintain separate items, therefore byte buffers have no overhead. All data is stored as a sequence of byte and any number of bytes can be sent or retrieved each time.

enumerator **RINGBUF_TYPE_MAX**

Hooks API

Header File

- components/esp_system/include/esp_freertos_hooks.h

Functions

esp_err_t **esp_register_freertos_idle_hook_for_cpu**(*esp_freertos_idle_cb_t* new_idle_cb, UBaseType_t cpuid)

Register a callback to be called from the specified core's idle hook. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

Warning: Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

Parameters
- **new_idle_cb** –**[in]** Callback to be called
- **cpuid** –**[in]** id of the core

Returns
- ESP_OK: Callback registered to the specified core's idle hook
- ESP_ERR_NO_MEM: No more space on the specified core's idle hook to register callback
- ESP_ERR_INVALID_ARG: cpuid is invalid

esp_err_t **esp_register_freertos_idle_hook**(*esp_freertos_idle_cb_t* new_idle_cb)

Register a callback to the idle hook of the core that calls this function. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

Warning: Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

Parameters **new_idle_cb** –**[in]** Callback to be called

Returns
- ESP_OK: Callback registered to the calling core's idle hook
- ESP_ERR_NO_MEM: No more space on the calling core's idle hook to register callback

esp_err_t **esp_register_freertos_tick_hook_for_cpu**(*esp_freertos_tick_cb_t* new_tick_cb, UBaseType_t cpuid)

Register a callback to be called from the specified core's tick hook.

Parameters
- **new_tick_cb** –**[in]** Callback to be called
- **cpuid** –**[in]** id of the core

Returns
- ESP_OK: Callback registered to specified core's tick hook
- ESP_ERR_NO_MEM: No more space on the specified core's tick hook to register the callback
- ESP_ERR_INVALID_ARG: cpuid is invalid

esp_err_t **esp_register_freertos_tick_hook**(*esp_freertos_tick_cb_t* new_tick_cb)

Register a callback to be called from the calling core's tick hook.

Parameters **new_tick_cb** –**[in]** Callback to be called
Returns
- ESP_OK: Callback registered to the calling core's tick hook
- ESP_ERR_NO_MEM: No more space on the calling core's tick hook to register the callback

void **esp_deregister_freertos_idle_hook_for_cpu**(*esp_freertos_idle_cb_t* old_idle_cb, UBaseType_t cpuid)

Unregister an idle callback from the idle hook of the specified core.

Parameters
- **old_idle_cb** –**[in]** Callback to be unregistered
- **cpuid** –**[in]** id of the core

void **esp_deregister_freertos_idle_hook**(*esp_freertos_idle_cb_t* old_idle_cb)

Unregister an idle callback. If the idle callback is registered to the idle hooks of both cores, the idle hook will be unregistered from both cores.

Parameters **old_idle_cb** –**[in]** Callback to be unregistered

void **esp_deregister_freertos_tick_hook_for_cpu**(*esp_freertos_tick_cb_t* old_tick_cb, UBaseType_t cpuid)

Unregister a tick callback from the tick hook of the specified core.

Parameters
- **old_tick_cb** –**[in]** Callback to be unregistered
- **cpuid** –**[in]** id of the core

void **esp_deregister_freertos_tick_hook**(*esp_freertos_tick_cb_t* old_tick_cb)

Unregister a tick callback. If the tick callback is registered to the tick hooks of both cores, the tick hook will be unregistered from both cores.

Parameters **old_tick_cb** –**[in]** Callback to be unregistered

Type Definitions

typedef bool (***esp_freertos_idle_cb_t**)(void)

typedef void (***esp_freertos_tick_cb_t**)(void)

2.10.13 Heap Memory Allocation

Stack and Heap

ESP-IDF applications use the common computer architecture patterns of *stack* (dynamic memory allocated by program control flow) and *heap* (dynamic memory allocated by function calls), as well as statically allocated memory (allocated at compile time).

Because ESP-IDF is a multi-threaded RTOS environment, each RTOS task has its own stack. By default, each of these stacks is allocated from the heap when the task is created. (See `xTaskCreateStatic()` for the alternative where stacks are statically allocated.)

Because ESP32 uses multiple types of RAM, it also contains multiple heaps with different capabilities. A capabilities-based memory allocator allows apps to make heap allocations for different purposes.

For most purposes, the standard libc `malloc()` and `free()` functions can be used for heap allocation without any special consideration.

However, in order to fully make use of all of the memory types and their characteristics, ESP-IDF also has a capabilities-based heap memory allocator. If you want to have memory with certain properties (for example, *DMA-Capable Memory* or executable-memory), you can create an OR-mask of the required capabilities and pass that to *`heap_caps_malloc()`*.

Memory Capabilities

The ESP32 contains multiple types of RAM:

- DRAM (Data RAM) is memory used to hold data. This is the most common kind of memory accessed as heap.
- IRAM (Instruction RAM) usually holds executable data only. If accessed as generic memory, all accesses must be *32-bit aligned*.
- D/IRAM is RAM which can be used as either Instruction or Data RAM.

For more details on these internal memory types, see *Memory Types*.

It's also possible to connect external SPI RAM to the ESP32 - *external RAM* can be integrated into the ESP32's memory map using the flash cache, and accessed similarly to DRAM.

DRAM uses capability `MALLOC_CAP_8BIT` (accessible in single byte reads and writes). To test the free DRAM heap size at runtime, call cpp:func:*heap_caps_get_free_size(MALLOC_CAP_8BIT)*.

When calling `malloc()`, the ESP-IDF `malloc()` implementation internally calls cpp:func:*heap_caps_malloc_default(size)*. This will allocate memory with capability `MALLOC_CAP_DEFAULT`, which is byte-addressable.

Because malloc uses the capabilities-based allocation system, memory allocated using *`heap_caps_malloc()`* can be freed by calling the standard `free()` function.

Available Heap

DRAM At startup, the DRAM heap contains all data memory which is not statically allocated by the app. Reducing statically allocated buffers will increase the amount of available free heap.

To find the amount of statically allocated memory, use the *idf.py size* command.

Note: Due to a technical limitation, the maximum statically allocated DRAM usage is 160KB. The remaining 160KB (for a total of 320KB of DRAM) can only be allocated at runtime as heap.

Note: At runtime, the available heap DRAM may be less than calculated at compile time, because at startup some memory is allocated from the heap before the FreeRTOS scheduler is started (including memory for the stacks of initial FreeRTOS tasks).

IRAM At startup, the IRAM heap contains all instruction memory which is not used by the app executable code.

The *idf.py size* command can be used to find the amount of IRAM used by the app.

D/IRAM Some memory in the ESP32 is available as either DRAM or IRAM. If memory is allocated from a D/IRAM region, the free heap size for both types of memory will decrease.

Heap Sizes At startup, all ESP-IDF apps log a summary of all heap addresses (and sizes) at level Info:

```
I (252) heap_init: Initializing. RAM available for dynamic allocation:
I (259) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (265) heap_init: At 3FFB2EC8 len 0002D138 (180 KiB): DRAM
I (272) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (278) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (284) heap_init: At 4008944C len 00016BB4 (90 KiB): IRAM
```

Finding available heap See *Heap Information*.

Special Capabilities

DMA-Capable Memory Use the `MALLOC_CAP_DMA` flag to allocate memory which is suitable for use with hardware DMA engines (for example SPI and I2S). This capability flag excludes any external PSRAM.

32-Bit Accessible Memory If a certain memory structure is only addressed in 32-bit units, for example an array of ints or pointers, it can be useful to allocate it with the `MALLOC_CAP_32BIT` flag. This also allows the allocator to give out IRAM memory; something which it can't do for a normal malloc() call. This can help to use all the available memory in the ESP32.

Memory allocated with `MALLOC_CAP_32BIT` can *only* be accessed via 32-bit reads and writes, any other type of access will generate a fatal LoadStoreError exception.

External SPI Memory When *external RAM* is enabled, external SPI RAM under 4MiB in size can be allocated using standard `malloc` calls, or via `heap_caps_malloc(MALLOC_CAP_SPIRAM)`, depending on configuration. See *Configuring External RAM* for more details.

To use the region above the 4MiB limit, you can use the *himem API*.

Thread Safety

Heap functions are thread safe, meaning they can be called from different tasks simultaneously without any limitations.

It is technically possible to call `malloc`, `free`, and related functions from interrupt handler (ISR) context (see *Calling heap related functions from ISR*). However this is not recommended, as heap function calls may delay other interrupts. It is strongly recommended to refactor applications so that any buffers used by an ISR are pre-allocated outside of the ISR. Support for calling heap functions from ISRs may be removed in a future update.

Calling heap related functions from ISR

The following functions from the heap component can be called form interrupt handler (ISR):

- *heap_caps_malloc()*
- heap_caps_malloc_default()
- heap_caps_realloc_default()
- *heap_caps_malloc_prefer()*
- *heap_caps_realloc_prefer()*
- *heap_caps_calloc_prefer()*
- *heap_caps_free()*
- *heap_caps_realloc()*
- *heap_caps_calloc()*
- *heap_caps_aligned_alloc()*
- *heap_caps_aligned_free()*

Note however this practice is strongly discouraged.

Heap Tracing & Debugging

The following features are documented on the *Heap Memory Debugging* page:

- *Heap Information* (free space, etc.)
- *Heap Corruption Detection*
- *Heap Tracing* (memory leak detection, monitoring, etc.)

Implementation Notes

Knowledge about the regions of memory in the chip comes from the "soc" component, which contains memory layout information for the chip, and the different capabilities of each region. Each region's capabilities are prioritised, so that (for example) dedicated DRAM and IRAM regions will be used for allocations ahead of the more versatile D/IRAM regions.

Each contiguous region of memory contains its own memory heap. The heaps are created using the *multi_heap* functionality. multi_heap allows any contiguous region of memory to be used as a heap.

The heap capabilities allocator uses knowledge of the memory regions to initialize each individual heap. Allocation functions in the heap capabilities API will find the most appropriate heap for the allocation (based on desired capabilities, available space, and preferences for each region's use) and then calling `multi_heap_malloc()` for the heap situated in that particular region.

Calling `free()` involves finding the particular heap corresponding to the freed address, and then calling `multi_heap_free()` on that particular multi_heap instance.

API Reference - Heap Allocation

Header File

- components/heap/include/esp_heap_caps.h

Functions

esp_err_t **heap_caps_register_failed_alloc_callback**(*esp_alloc_failed_hook_t* callback)

registers a callback function to be invoked if a memory allocation operation fails

Parameters **callback** –caller defined callback to be invoked
Returns ESP_OK if callback was registered.

void ***heap_caps_malloc**(size_t size, uint32_t caps)

Allocate a chunk of memory which has the given capabilities.

Equivalent semantics to libc malloc(), for capability-aware memory.

Parameters
- **size** –Size, in bytes, of the amount of memory to allocate
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

Returns A pointer to the memory allocated on success, NULL on failure

void **heap_caps_free**(void *ptr)

Free memory previously allocated via heap_caps_malloc() or heap_caps_realloc().

Equivalent semantics to libc free(), for capability-aware memory.

In IDF, `free(p)` is equivalent to `heap_caps_free(p)`.

Parameters **ptr** –Pointer to memory previously returned from heap_caps_malloc() or heap_caps_realloc(). Can be NULL.

void ***heap_caps_realloc**(void *ptr, size_t size, uint32_t caps)

Reallocate memory previously allocated via heap_caps_malloc() or heap_caps_realloc().

Equivalent semantics to libc realloc(), for capability-aware memory.

In IDF, `realloc(p, s)` is equivalent to `heap_caps_realloc(p, s, MALLOC_CAP_8BIT)`.

'caps' parameter can be different to the capabilities that any original 'ptr' was allocated with. In this way, realloc can be used to "move" a buffer if necessary to ensure it meets a new set of capabilities.

Parameters
- **ptr** –Pointer to previously allocated memory, or NULL for a new allocation.
- **size** –Size of the new buffer requested, or 0 to free the buffer.
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory desired for the new allocation.

Returns Pointer to a new buffer of size 'size' with capabilities 'caps', or NULL if allocation failed.

void ***heap_caps_aligned_alloc**(size_t alignment, size_t size, uint32_t caps)

Allocate an aligned chunk of memory which has the given capabilities.

Equivalent semantics to libc aligned_alloc(), for capability-aware memory.

Parameters
- **alignment** –How the pointer received needs to be aligned must be a power of two
- **size** –Size, in bytes, of the amount of memory to allocate
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

Returns A pointer to the memory allocated on success, NULL on failure

void **heap_caps_aligned_free**(void *ptr)

Used to deallocate memory previously allocated with heap_caps_aligned_alloc.

Note: This function is deprecated, please consider using heap_caps_free() instead

---

Parameters **ptr** –Pointer to the memory allocated

void ***heap_caps_aligned_calloc**(size_t alignment, size_t n, size_t size, uint32_t caps)

Allocate an aligned chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Parameters
- **alignment** –How the pointer received needs to be aligned must be a power of two
- **n** –Number of continuing chunks of memory to allocate
- **size** –Size, in bytes, of a chunk of memory to allocate
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

Returns A pointer to the memory allocated on success, NULL on failure

void ***heap_caps_calloc**(size_t n, size_t size, uint32_t caps)

Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to libc calloc(), for capability-aware memory.

In IDF, `calloc(p)` is equivalent to `heap_caps_calloc(p, MALLOC_CAP_8BIT)`.

Parameters
- **n** –Number of continuing chunks of memory to allocate
- **size** –Size, in bytes, of a chunk of memory to allocate
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

Returns A pointer to the memory allocated on success, NULL on failure

size_t **heap_caps_get_total_size**(uint32_t caps)

Get the total size of all the regions that have the given capabilities.

This function takes all regions capable of having the given capabilities allocated in them and adds up the total space they have.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory
Returns total size in bytes

size_t **heap_caps_get_free_size**(uint32_t caps)

Get the total free size of all the regions that have the given capabilities.

This function takes all regions capable of having the given capabilities allocated in them and adds up the free space they have.

Note: Note that because of heap fragmentation it is probably not possible to allocate a single block of memory of this size. Use heap_caps_get_largest_free_block() for this purpose.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory
Returns Amount of free bytes in the regions

size_t **heap_caps_get_minimum_free_size**(uint32_t caps)

Get the total minimum free memory of all regions with the given capabilities.

This adds all the low watermarks of the regions capable of delivering the memory with the given capabilities.

Note: Note the result may be less than the global all-time minimum available heap of this kind, as "low watermarks" are tracked per-region. Individual regions' heaps may have reached their "low watermarks" at different points in time. However, this result still gives a "worst case" indication for all-time minimum free heap.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory
Returns Amount of free bytes in the regions

size_t **heap_caps_get_largest_free_block**(uint32_t caps)

Get the largest free block of memory able to be allocated with the given capabilities.

Returns the largest value of `s` for which `heap_caps_malloc(s, caps)` will succeed.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory
Returns Size of the largest free block in bytes.

void **heap_caps_get_info**(*multi_heap_info_t* *info, uint32_t caps)

Get heap info for all regions with the given capabilities.

Calls multi_heap_info() on all heaps which share the given capabilities. The information returned is an aggregate across all matching heaps. The meanings of fields are the same as defined for *multi_heap_info_t*, except that `minimum_free_bytes` has the same caveats described in heap_caps_get_minimum_free_size().

Parameters

- **info** –Pointer to a structure which will be filled with relevant heap metadata.
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

void **heap_caps_print_heap_info**(uint32_t caps)

Print a summary of all memory with the given capabilities.

Calls multi_heap_info on all heaps which share the given capabilities, and prints a two-line summary for each, then a total summary.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

bool **heap_caps_check_integrity_all**(bool print_errors)

Check integrity of all heap memory in the system.

Calls multi_heap_check on all heaps. Optionally print errors if heaps are corrupt.

Calling this function is equivalent to calling heap_caps_check_integrity with the caps argument set to MALLOC_CAP_INVALID.

Parameters **print_errors** –Print specific errors if heap corruption is found.
Returns True if all heaps are valid, False if at least one heap is corrupt.

bool **heap_caps_check_integrity**(uint32_t caps, bool print_errors)

Check integrity of all heaps with the given capabilities.

Calls multi_heap_check on all heaps which share the given capabilities. Optionally print errors if the heaps are corrupt.

See also heap_caps_check_integrity_all to check all heap memory in the system and heap_caps_check_integrity_addr to check memory around a single address.

Parameters
- **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory
- **print_errors** –Print specific errors if heap corruption is found.

Returns True if all heaps are valid, False if at least one heap is corrupt.

bool **heap_caps_check_integrity_addr**(intptr_t addr, bool print_errors)

Check integrity of heap memory around a given address.

This function can be used to check the integrity of a single region of heap memory, which contains the given address.

This can be useful if debugging heap integrity for corruption at a known address, as it has a lower overhead than checking all heap regions. Note that if the corrupt address moves around between runs (due to timing or other factors) then this approach won't work, and you should call heap_caps_check_integrity or heap_caps_check_integrity_all instead.

Note: The entire heap region around the address is checked, not only the adjacent heap blocks.

Parameters
- **addr** –Address in memory. Check for corruption in region containing this address.
- **print_errors** –Print specific errors if heap corruption is found.

Returns True if the heap containing the specified address is valid, False if at least one heap is corrupt or the address doesn't belong to a heap region.

void **heap_caps_malloc_extmem_enable**(size_t limit)

Enable malloc() in external memory and set limit below which malloc() attempts are placed in internal memory.

When external memory is in use, the allocation strategy is to initially try to satisfy smaller allocation requests with internal memory and larger requests with external memory. This sets the limit between the two, as well as generally enabling allocation in external memory.

Parameters **limit** –Limit, in bytes.

void ***heap_caps_malloc_prefer**(size_t size, size_t num, ...)

Allocate a chunk of memory as preference in decreasing order.

Attention The variable parameters are bitwise OR of MALLOC_CAP_* flags indicating the type of memory. This API prefers to allocate memory with the first parameter. If failed, allocate memory with the next parameter. It will try in this order until allocating a chunk of memory successfully or fail to allocate memories with any of the parameters.

Parameters
- **size** –Size, in bytes, of the amount of memory to allocate
- **num** –Number of variable parameters

Returns A pointer to the memory allocated on success, NULL on failure

void ***heap_caps_realloc_prefer**(void *ptr, size_t size, size_t num, ...)

Reallocate a chunk of memory as preference in decreasing order.

Parameters
- **ptr** –Pointer to previously allocated memory, or NULL for a new allocation.
- **size** –Size of the new buffer requested, or 0 to free the buffer.
- **num** –Number of variable paramters

Returns Pointer to a new buffer of size ‘size’, or NULL if allocation failed.

void ***heap_caps_calloc_prefer**(size_t n, size_t size, size_t num, ...)

Allocate a chunk of memory as preference in decreasing order.

Parameters
- **n** –Number of continuing chunks of memory to allocate
- **size** –Size, in bytes, of a chunk of memory to allocate
- **num** –Number of variable paramters

Returns A pointer to the memory allocated on success, NULL on failure

void **heap_caps_dump**(uint32_t caps)

Dump the full structure of all heaps with matching capabilities.

Prints a large amount of output to serial (because of locking limitations, the output bypasses stdout/stderr). For each (variable sized) block in each matching heap, the following output is printed on a single line:

- Block address (the data buffer returned by malloc is 4 bytes after this if heap debugging is set to Basic, or 8 bytes otherwise).
- Data size (the data size may be larger than the size requested by malloc, either due to heap fragmentation or because of heap debugging level).
- Address of next block in the heap.
- If the block is free, the address of the next free block is also printed.

Parameters **caps** –Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

void **heap_caps_dump_all**(void)

Dump the full structure of all heaps.

Covers all registered heaps. Prints a large amount of output to serial.

Output is the same as for heap_caps_dump.

size_t **heap_caps_get_allocated_size**(void *ptr)

Return the size that a particular pointer was allocated with.

Note: The app will crash with an assertion failure if the pointer is not valid.

Parameters **ptr** –Pointer to currently allocated heap memory. Must be a pointer value previously returned by heap_caps_malloc, malloc, calloc, etc. and not yet freed.

Returns Size of the memory allocated at this block.

Macros

MALLOC_CAP_EXEC

Flags to indicate the capabilities of the various memory systems.

Memory must be able to run executable code

MALLOC_CAP_32BIT

Memory must allow for aligned 32-bit data accesses.

MALLOC_CAP_8BIT

Memory must allow for 8/16/…-bit data accesses.

MALLOC_CAP_DMA

Memory must be able to accessed by DMA.

MALLOC_CAP_PID2

Memory must be mapped to PID2 memory space (PIDs are not currently used)

MALLOC_CAP_PID3

Memory must be mapped to PID3 memory space (PIDs are not currently used)

MALLOC_CAP_PID4

Memory must be mapped to PID4 memory space (PIDs are not currently used)

MALLOC_CAP_PID5

Memory must be mapped to PID5 memory space (PIDs are not currently used)

MALLOC_CAP_PID6

Memory must be mapped to PID6 memory space (PIDs are not currently used)

MALLOC_CAP_PID7

Memory must be mapped to PID7 memory space (PIDs are not currently used)

MALLOC_CAP_SPIRAM

Memory must be in SPI RAM.

MALLOC_CAP_INTERNAL

Memory must be internal; specifically it should not disappear when flash/spiram cache is switched off.

MALLOC_CAP_DEFAULT

Memory can be returned in a non-capability-specific memory allocation (e.g. malloc(), calloc()) call.

MALLOC_CAP_IRAM_8BIT

Memory must be in IRAM and allow unaligned access.

MALLOC_CAP_RETENTION

Memory must be able to accessed by retention DMA.

MALLOC_CAP_RTCRAM

Memory must be in RTC fast memory.

MALLOC_CAP_INVALID

Memory can't be used / list end marker.

Type Definitions

typedef void (***esp_alloc_failed_hook_t**)(size_t size, uint32_t caps, const char *function_name)

callback called when an allocation operation fails, if registered

Param size in bytes of failed allocation
Param caps capabilities requested of failed allocation
Param function_name function which generated the failure

API Reference - Initialisation

Header File

- components/heap/include/esp_heap_caps_init.h

Functions

void **heap_caps_init**(void)

Initialize the capability-aware heap allocator.

This is called once in the IDF startup code. Do not call it at other times.

void **heap_caps_enable_nonos_stack_heaps**(void)

Enable heap(s) in memory regions where the startup stacks are located.

On startup, the pro/app CPUs have a certain memory region they use as stack, so we cannot do allocations in the regions these stack frames are. When FreeRTOS is completely started, they do not use that memory anymore and heap(s) there can be enabled.

esp_err_t **heap_caps_add_region**(intptr_t start, intptr_t end)

Add a region of memory to the collection of heaps at runtime.

Most memory regions are defined in soc_memory_layout.c for the SoC, and are registered via heap_caps_init(). Some regions can't be used immediately and are later enabled via heap_caps_enable_nonos_stack_heaps().

Call this function to add a region of memory to the heap at some later time.

This function does not consider any of the "reserved" regions or other data in soc_memory_layout, caller needs to consider this themselves.

All memory within the region specified by start & end parameters must be otherwise unused.

The capabilities of the newly registered memory will be determined by the start address, as looked up in the regions specified in soc_memory_layout.c.

Use heap_caps_add_region_with_caps() to register a region with custom capabilities.

Note: Please refer to following example for memory regions allowed for addition to heap based on an existing region (address range for demonstration purpose only):

```
Existing region: 0x1000 <-> 0x3000
New region:      0x1000 <-> 0x3000 (Allowed)
New region:      0x1000 <-> 0x2000 (Allowed)
New region:      0x0000 <-> 0x1000 (Allowed)
New region:      0x3000 <-> 0x4000 (Allowed)
New region:      0x0000 <-> 0x2000 (NOT Allowed)
New region:      0x0000 <-> 0x4000 (NOT Allowed)
New region:      0x1000 <-> 0x4000 (NOT Allowed)
New region:      0x2000 <-> 0x4000 (NOT Allowed)
```

Parameters

- **start** –Start address of new region.
- **end** –End address of new region.

Returns ESP_OK on success, ESP_ERR_INVALID_ARG if a parameter is invalid, ESP_ERR_NOT_FOUND if the specified start address doesn't reside in a known region, or any error returned by heap_caps_add_region_with_caps().

esp_err_t **heap_caps_add_region_with_caps**(const uint32_t caps[], intptr_t start, intptr_t end)

Add a region of memory to the collection of heaps at runtime, with custom capabilities.

Similar to heap_caps_add_region(), only custom memory capabilities are specified by the caller.

Note: Please refer to following example for memory regions allowed for addition to heap based on an existing region (address range for demonstration purpose only):

```
Existing region: 0x1000 <-> 0x3000
New region:      0x1000 <-> 0x3000 (Allowed)
New region:      0x1000 <-> 0x2000 (Allowed)
New region:      0x0000 <-> 0x1000 (Allowed)
New region:      0x3000 <-> 0x4000 (Allowed)
New region:      0x0000 <-> 0x2000 (NOT Allowed)
New region:      0x0000 <-> 0x4000 (NOT Allowed)
New region:      0x1000 <-> 0x4000 (NOT Allowed)
New region:      0x2000 <-> 0x4000 (NOT Allowed)
```

Parameters

- **caps** –Ordered array of capability masks for the new region, in order of priority. Must have length SOC_MEMORY_TYPE_NO_PRIOS. Does not need to remain valid after the call returns.
- **start** –Start address of new region.
- **end** –End address of new region.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if a parameter is invalid
- ESP_ERR_NO_MEM if no memory to register new heap.
- ESP_ERR_INVALID_SIZE if the memory region is too small to fit a heap
- ESP_FAIL if region overlaps the start and/or end of an existing region

API Reference - Multi Heap API

(Note: The multi heap API is used internally by the heap capabilities allocator. Most IDF programs will never need to call this API directly.)

Header File

- components/heap/include/multi_heap.h

Functions

void ***multi_heap_aligned_alloc**(*multi_heap_handle_t* heap, size_t size, size_t alignment)

allocate a chunk of memory with specific alignment

Parameters

- **heap** –Handle to a registered heap.
- **size** –size in bytes of memory chunk

- **alignment** –how the memory must be aligned

Returns pointer to the memory allocated, NULL on failure

void ***multi_heap_malloc**(*multi_heap_handle_t* heap, size_t size)

malloc() a buffer in a given heap

Semantics are the same as standard malloc(), only the returned buffer will be allocated in the specified heap.

Parameters

- **heap** –Handle to a registered heap.
- **size** –Size of desired buffer.

Returns Pointer to new memory, or NULL if allocation fails.

void **multi_heap_aligned_free**(*multi_heap_handle_t* heap, void *p)

free() a buffer aligned in a given heap.

Note: This function is deprecated, consider using multi_heap_free() instead

---

Parameters

- **heap** –Handle to a registered heap.
- **p** –NULL, or a pointer previously returned from multi_heap_aligned_alloc() for the same heap.

void **multi_heap_free**(*multi_heap_handle_t* heap, void *p)

free() a buffer in a given heap.

Semantics are the same as standard free(), only the argument ‘p’ must be NULL or have been allocated in the specified heap.

Parameters

- **heap** –Handle to a registered heap.
- **p** –NULL, or a pointer previously returned from multi_heap_malloc() or multi_heap_realloc() for the same heap.

void ***multi_heap_realloc**(*multi_heap_handle_t* heap, void *p, size_t size)

realloc() a buffer in a given heap.

Semantics are the same as standard realloc(), only the argument ‘p’ must be NULL or have been allocated in the specified heap.

Parameters

- **heap** –Handle to a registered heap.
- **p** –NULL, or a pointer previously returned from multi_heap_malloc() or multi_heap_realloc() for the same heap.
- **size** –Desired new size for buffer.

Returns New buffer of ‘size’ containing contents of ‘p’, or NULL if reallocation failed.

size_t **multi_heap_get_allocated_size**(*multi_heap_handle_t* heap, void *p)

Return the size that a particular pointer was allocated with.

Parameters

- **heap** –Handle to a registered heap.
- **p** –Pointer, must have been previously returned from multi_heap_malloc() or multi_heap_realloc() for the same heap.

Returns Size of the memory allocated at this block. May be more than the original size argument, due to padding and minimum block sizes.

multi_heap_handle_t **multi_heap_register**(void *start, size_t size)

Register a new heap for use.

This function initialises a heap at the specified address, and returns a handle for future heap operations.

There is no equivalent function for deregistering a heap - if all blocks in the heap are free, you can immediately start using the memory for other purposes.

Parameters
- **`start`** –Start address of the memory to use for a new heap.
- **`size`** –Size (in bytes) of the new heap.

Returns Handle of a new heap ready for use, or NULL if the heap region was too small to be initialised.

void **`multi_heap_set_lock`**(*multi_heap_handle_t* heap, void *lock)

Associate a private lock pointer with a heap.

The lock argument is supplied to the MULTI_HEAP_LOCK() and MULTI_HEAP_UNLOCK() macros, defined in multi_heap_platform.h.

The lock in question must be recursive.

When the heap is first registered, the associated lock is NULL.

Parameters
- **`heap`** –Handle to a registered heap.
- **`lock`** –Optional pointer to a locking structure to associate with this heap.

void **`multi_heap_dump`**(*multi_heap_handle_t* heap)

Dump heap information to stdout.

For debugging purposes, this function dumps information about every block in the heap to stdout.

Parameters **`heap`** –Handle to a registered heap.

bool **`multi_heap_check`**(*multi_heap_handle_t* heap, bool print_errors)

Check heap integrity.

Walks the heap and checks all heap data structures are valid. If any errors are detected, an error-specific message can be optionally printed to stderr. Print behaviour can be overridden at compile time by defining MULTI_CHECK_FAIL_PRINTF in multi_heap_platform.h.

Note: This function is not thread-safe as it sets a global variable with the value of print_errors.

Parameters
- **`heap`** –Handle to a registered heap.
- **`print_errors`** –If true, errors will be printed to stderr.

Returns true if heap is valid, false otherwise.

size_t **`multi_heap_free_size`**(*multi_heap_handle_t* heap)

Return free heap size.

Returns the number of bytes available in the heap.

Equivalent to the total_free_bytes member returned by multi_heap_get_heap_info().

Note that the heap may be fragmented, so the actual maximum size for a single malloc() may be lower. To know this size, see the largest_free_block member returned by multi_heap_get_heap_info().

Parameters **`heap`** –Handle to a registered heap.
Returns Number of free bytes.

size_t **`multi_heap_minimum_free_size`**(*multi_heap_handle_t* heap)

Return the lifetime minimum free heap size.

Equivalent to the minimum_free_bytes member returned by multi_heap_get_info().

Returns the lifetime "low watermark" of possible values returned from multi_free_heap_size(), for the specified heap.

Parameters **heap** –Handle to a registered heap.
Returns Number of free bytes.

void **multi_heap_get_info**(*multi_heap_handle_t* heap, *multi_heap_info_t* *info)

Return metadata about a given heap.

Fills a *multi_heap_info_t* structure with information about the specified heap.

Parameters
- **heap** –Handle to a registered heap.
- **info** –Pointer to a structure to fill with heap metadata.

Structures

struct **multi_heap_info_t**

Structure to access heap metadata via multi_heap_get_info.

Public Members

size_t **total_free_bytes**

Total free bytes in the heap. Equivalent to multi_free_heap_size().

size_t **total_allocated_bytes**

Total bytes allocated to data in the heap.

size_t **largest_free_block**

Size of the largest free block in the heap. This is the largest malloc-able size.

size_t **minimum_free_bytes**

Lifetime minimum free heap size. Equivalent to multi_minimum_free_heap_size().

size_t **allocated_blocks**

Number of (variable size) blocks allocated in the heap.

size_t **free_blocks**

Number of (variable size) free blocks in the heap.

size_t **total_blocks**

Total number of (variable size) blocks in the heap.

Type Definitions

typedef struct multi_heap_info *__multi_heap_handle_t__

Opaque handle to a registered heap.

2.10.14 Heap Memory Debugging

Overview

ESP-IDF integrates tools for requesting *heap information*, *detecting heap corruption*, and *tracing memory leaks*. These can help track down memory-related bugs.

For general information about the heap memory allocator, see the *Heap Memory Allocation* page.

Heap Information

To obtain information about the state of the heap:

- `xPortGetFreeHeapSize()` is a FreeRTOS function which returns the number of free bytes in the (data memory) heap. This is equivalent to calling `heap_caps_get_free_size(MALLOC_CAP_8BIT)`.
- *`heap_caps_get_free_size()`* can also be used to return the current free memory for different memory capabilities.
- *`heap_caps_get_largest_free_block()`* can be used to return the largest free block in the heap. This is the largest single allocation which is currently possible. Tracking this value and comparing to total free heap allows you to detect heap fragmentation.
- `xPortGetMinimumEverFreeHeapSize()` and the related *`heap_caps_get_minimum_free_size()`* can be used to track the heap "low watermark" since boot.
- *`heap_caps_get_info()`* returns a *`multi_heap_info_t`* structure which contains the information from the above functions, plus some additional heap-specific data (number of allocations, etc.).
- *`heap_caps_print_heap_info()`* prints a summary to stdout of the information returned by *`heap_caps_get_info()`*.
- *`heap_caps_dump()`* and *`heap_caps_dump_all()`* will output detailed information about the structure of each block in the heap. Note that this can be large amount of output.

Heap Corruption Detection

Heap corruption detection allows you to detect various types of heap memory errors:

- Out of bounds writes & buffer overflow.
- Writes to freed memory.
- Reads from freed or uninitialized memory,

Assertions The heap implementation (`multi_heap.c`, etc.) includes a lot of assertions which will fail if the heap memory is corrupted. To detect heap corruption most effectively, ensure that assertions are enabled in the project configuration menu under `Compiler options` -> *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*.

If a heap integrity assertion fails, a line will be printed like `CORRUPT HEAP: multi_heap.c:225 detected at 0x3ffbb71c`. The memory address which is printed is the address of the heap structure which has corrupt content.

It's also possible to manually check heap integrity by calling *`heap_caps_check_integrity_all()`* or related functions. This function checks all of requested heap memory for integrity, and can be used even if assertions are disabled. If the integrity check prints an error, it will also contain the address(es) of corrupt heap structures.

Memory Allocation Failed Hook Users can use *`heap_caps_register_failed_alloc_callback()`* to register a callback that will be invoked every time an allocation operation fails.

Additionally, users can enable the generation of a system abort if an allocation operation fails by following the steps below: - In the project configuration menu, navigate to `Component config` -> `Heap Memory Debugging` and select `Abort if memory allocation fails` option (see *CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS*).

The example below shows how to register an allocation failure callback:

```
#include "esp_heap_caps.h"

void heap_caps_alloc_failed_hook(size_t requested_size, uint32_t caps, const char␣
↪*function_name)
{
  printf("%s was called but failed to allocate %d bytes with 0x%X capabilities. \n
↪",function_name, requested_size, caps);
}
```

(continues on next page)

(continued from previous page)

```
void app_main()
{
    ...
    esp_err_t error = heap_caps_register_failed_alloc_callback(heap_caps_alloc_
↪failed_hook);
    ...
    void *ptr = heap_caps_malloc(allocation_size, MALLOC_CAP_DEFAULT);
    ...
}
```

Finding Heap Corruption Memory corruption can be one of the hardest classes of bugs to find and fix, as one area of memory can be corrupted from a totally different place. Some tips:

- A crash with a `CORRUPT HEAP:` message will usually include a stack trace, but this stack trace is rarely useful. The crash is the symptom of memory corruption when the system realises the heap is corrupt, but usually the corruption happened elsewhere and earlier in time.
- Increasing the Heap memory debugging *Configuration* level to "Light impact" or "Comprehensive" can give you a more accurate message with the first corrupt memory address.
- Adding regular calls to `heap_caps_check_integrity_all()` or `heap_caps_check_integrity_addr()` in your code will help you pin down the exact time that the corruption happened. You can move these checks around to "close in on" the section of code that corrupted the heap.
- Based on the memory address which is being corrupted, you can use *JTAG debugging* to set a watchpoint on this address and have the CPU halt when it is written to.
- If you don't have JTAG, but you do know roughly when the corruption happens, then you can set a watchpoint in software just beforehand via `esp_cpu_set_watchpoint()`. A fatal exception will occur when the watchpoint triggers. The following is an example of how to use the function - `esp_cpu_set_watchpoint(0, (void *)addr, 4, ESP_WATCHPOINT_STORE)`. Note that watchpoints are per-CPU and are set on the current running CPU only, so if you don't know which CPU is corrupting memory then you will need to call this function on both CPUs.
- For buffer overflows, *heap tracing* in `HEAP_TRACE_ALL` mode lets you see which callers are allocating which addresses from the heap. See *Heap Tracing To Find Heap Corruption* for more details. If you can find the function which allocates memory with an address immediately before the address which is corrupted, this will probably be the function which overflows the buffer.
- Calling `heap_caps_dump()` or `heap_caps_dump_all()` can give an indication of what heap blocks are surrounding the corrupted region and may have overflowed/underflowed/etc.

Configuration Temporarily increasing the heap corruption detection level can give more detailed information about heap corruption errors.

In the project configuration menu, under `Component config` there is a menu `Heap memory debugging`. The setting *CONFIG_HEAP_CORRUPTION_DETECTION* can be set to one of three levels:

Basic (no poisoning) This is the default level. No special heap corruption features are enabled, but provided assertions are enabled (the default configuration) then a heap corruption error will be printed if any of the heap's internal data structures appear overwritten or corrupted. This usually indicates a buffer overrun or out of bounds write.

If assertions are enabled, an assertion will also trigger if a double-free occurs (the same memory is freed twice).

Calling `heap_caps_check_integrity()` in Basic mode will check the integrity of all heap structures, and print errors if any appear to be corrupted.

Light Impact At this level, heap memory is additionally "poisoned" with head and tail "canary bytes" before and after each block which is allocated. If an application writes outside the bounds of allocated buffers, the canary bytes will be corrupted and the integrity check will fail.

The head canary word is 0xABBA1234 (3412BAAB in byte order), and the tail canary word is 0xBAAD5678 (7856ADBA in byte order).

"Basic" heap corruption checks can also detect most out of bounds writes, but this setting is more precise as even a single byte overrun can be detected. With Basic heap checks, the number of overrun bytes before a failure is detected will depend on the properties of the heap.

Enabling "Light Impact" checking increases memory usage, each individual allocation will use 9 to 12 additional bytes of memory (depending on alignment).

Each time `free()` is called in Light Impact mode, the head and tail canary bytes of the buffer being freed are checked against the expected values.

When *`heap_caps_check_integrity()`* is called, all allocated blocks of heap memory have their canary bytes checked against the expected values.

In both cases, the check is that the first 4 bytes of an allocated block (before the buffer returned to the user) should be the word 0xABBA1234. Then the last 4 bytes of the allocated block (after the buffer returned to the user) should be the word 0xBAAD5678.

Different values usually indicate buffer underrun or overrun, respectively.

Comprehensive This level incorporates the "light impact" detection features plus additional checks for uninitialised-access and use-after-free bugs. In this mode, all freshly allocated memory is filled with the pattern 0xCE, and all freed memory is filled with the pattern 0xFE.

Enabling "Comprehensive" detection has a substantial runtime performance impact (as all memory needs to be set to the allocation patterns each time a malloc/free completes, and the memory also needs to be checked each time.) However, it allows easier detection of memory corruption bugs which are much more subtle to find otherwise. It is recommended to only enable this mode when debugging, not in production.

Crashes in Comprehensive Mode If an application crashes reading/writing an address related to 0xCECECECE in Comprehensive mode, this indicates it has read uninitialized memory. The application should be changed to either use calloc() (which zeroes memory), or initialize the memory before using it. The value 0xCECECECE may also be seen in stack-allocated automatic variables, because in IDF most task stacks are originally allocated from the heap and in C stack memory is uninitialized by default.

If an application crashes and the exception register dump indicates that some addresses or values were 0xFEFEFEFE, this indicates it is reading heap memory after it has been freed (a "use after free bug".) The application should be changed to not access heap memory after it has been freed.

If a call to malloc() or realloc() causes a crash because it expected to find the pattern 0xFEFEFEFE in free memory and a different pattern was found, then this indicates the app has a use-after-free bug where it is writing to memory which has already been freed.

Manual Heap Checks in Comprehensive Mode Calls to *`heap_caps_check_integrity()`* may print errors relating to 0xFEFEFEFE, 0xABBA1234 or 0xBAAD5678. In each case the checker is expecting to find a given pattern, and will error out if this is not found:

- For free heap blocks, the checker expects to find all bytes set to 0xFE. Any other values indicate a use-after-free bug where free memory has been incorrectly overwritten.
- For allocated heap blocks, the behaviour is the same as for *Light Impact* mode. The canary bytes 0xABBA1234 and 0xBAAD5678 are checked at the head and tail of each allocated buffer, and any variation indicates a buffer overrun/underrun.

Heap Task Tracking

Heap Task Tracking can be used to get per task info for heap memory allocation. Application has to specify the heap capabilities for which the heap allocation is to be tracked.

Example code is provided in system/heap_task_tracking

Heap Tracing

Heap Tracing allows tracing of code which allocates/frees memory. Two tracing modes are supported:

- Standalone. In this mode trace data are kept on-board, so the size of gathered information is limited by the buffer assigned for that purposes. Analysis is done by the on-board code. There are a couple of APIs available for accessing and dumping collected info.
- Host-based. This mode does not have the limitation of the standalone mode, because trace data are sent to the host over JTAG connection using app_trace library. Later on they can be analysed using special tools.

Heap tracing can perform two functions:

- Leak checking: find memory which is allocated and never freed.
- Heap use analysis: show all functions that are allocating/freeing memory while the trace is running.

How To Diagnose Memory Leaks If you suspect a memory leak, the first step is to figure out which part of the program is leaking memory. Use the `xPortGetFreeHeapSize()`, *`heap_caps_get_free_size()`*, or *related functions* to track memory use over the life of the application. Try to narrow the leak down to a single function or sequence of functions where free memory always decreases and never recovers.

Standalone Mode Once you've identified the code which you think is leaking:

- In the project configuration menu, navigate to `Component settings` -> `Heap Memory Debugging` -> `Heap tracing` and select `Standalone` option (see *CONFIG_HEAP_TRACING_DEST*).
- Call the function *`heap_trace_init_standalone()`* early in the program, to register a buffer which can be used to record the memory trace.
- Call the function *`heap_trace_start()`* to begin recording all mallocs/frees in the system. Call this immediately before the piece of code which you suspect is leaking memory.
- Call the function *`heap_trace_stop()`* to stop the trace once the suspect piece of code has finished executing.
- Call the function *`heap_trace_dump()`* to dump the results of the heap trace.

An example:

```
#include "esp_heap_trace.h"

#define NUM_RECORDS 100
static heap_trace_record_t trace_record[NUM_RECORDS]; // This buffer must be in␣
↪internal RAM

...

void app_main()
{
    ...
    ESP_ERROR_CHECK( heap_trace_init_standalone(trace_record, NUM_RECORDS) );
    ...
}

void some_function()
{
    ESP_ERROR_CHECK( heap_trace_start(HEAP_TRACE_LEAKS) );

    do_something_you_suspect_is_leaking();

    ESP_ERROR_CHECK( heap_trace_stop() );
    heap_trace_dump();
    ...
}
```

The output from the heap trace will look something like this:

```
2 allocations trace (100 entry buffer)
32 bytes (@ 0x3ffaf214) allocated CPU 0 ccount 0x2e9b7384 caller␣
↪0x400d276d:0x400d27c1
0x400d276d: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./
↪blink.c:27

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

8 bytes (@ 0x3ffaf804) allocated CPU 0 ccount 0x2e9b79c0 caller␣
↪0x400d2776:0x400d27c1
0x400d2776: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./
↪blink.c:29

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

40 bytes 'leaked' in trace (2 allocations)
total allocations 2 total frees 0
```

(Above example output is using *IDF Monitor* to automatically decode PC addresses to their source files & line number.)

The first line indicates how many allocation entries are in the buffer, compared to its total size.

In `HEAP_TRACE_LEAKS` mode, for each traced memory allocation which has not already been freed a line is printed with:

- `XX bytes` is the number of bytes allocated
- `@ 0x...` is the heap address returned from malloc/calloc.
- `CPU x` is the CPU (0 or 1) running when the allocation was made.
- `ccount 0x...` is the CCOUNT (CPU cycle count) register value when the allocation was mode. Is different for CPU 0 vs CPU 1.
- `caller 0x...` gives the call stack of the call to malloc()/free(), as a list of PC addresses. These can be decoded to source files and line numbers, as shown above.

The depth of the call stack recorded for each trace entry can be configured in the project configuration menu, under `Heap Memory Debugging` -> `Enable heap tracing` -> `Heap tracing stack depth`. Up to 10 stack frames can be recorded for each allocation (the default is 2). Each additional stack frame increases the memory usage of each `heap_trace_record_t` record by eight bytes.

Finally, the total number of 'leaked' bytes (bytes allocated but not freed while trace was running) is printed, and the total number of allocations this represents.

A warning will be printed if the trace buffer was not large enough to hold all the allocations which happened. If you see this warning, consider either shortening the tracing period or increasing the number of records in the trace buffer.

Host-Based Mode Once you've identified the code which you think is leaking:

- In the project configuration menu, navigate to `Component settings` -> `Heap Memory Debugging` -> *CONFIG_HEAP_TRACING_DEST* and select `Host-Based`.
- In the project configuration menu, navigate to `Component settings` -> `Application Level Tracing` -> *CONFIG_APPTRACE_DESTINATION1* and select `Trace memory`.
- In the project configuration menu, navigate to `Component settings` -> `Application Level Tracing` -> `FreeRTOS SystemView Tracing` and enable *CONFIG_APPTRACE_SV_ENABLE*.
- Call the function *heap_trace_init_tohost()* early in the program, to initialize JTAG heap tracing module.
- Call the function *heap_trace_start()* to begin recording all mallocs/frees in the system. Call this immediately before the piece of code which you suspect is leaking memory. In host-based mode, the argument to this function is ignored, and the heap tracing module behaves like `HEAP_TRACE_ALL` was passed: all allocations and deallocations are sent to the host.
- Call the function *heap_trace_stop()* to stop the trace once the suspect piece of code has finished executing.

An example:

```
#include "esp_heap_trace.h"

...

void app_main()
{
    ...
    ESP_ERROR_CHECK( heap_trace_init_tohost() );
    ...
}

void some_function()
{
    ESP_ERROR_CHECK( heap_trace_start(HEAP_TRACE_LEAKS) );

    do_something_you_suspect_is_leaking();

    ESP_ERROR_CHECK( heap_trace_stop() );
    ...
}
```

To gather and analyse heap trace do the following on the host:

1. Build the program and download it to the target as described in *Getting Started Guide*.
2. Run OpenOCD (see *JTAG Debugging*).

Note: In order to use this feature you need OpenOCD version *v0.10.0-esp32-20181105* or later.

---

3. You can use GDB to start and/or stop tracing automatically. To do this you need to prepare special `gdbinit` file:

```
target remote :3333

mon reset halt
flushregs

tb heap_trace_start
commands
mon esp sysview start file:///tmp/heap.svdat
c
end

tb heap_trace_stop
commands
mon esp sysview stop
end

c
```

Using this file GDB will connect to the target, reset it, and start tracing when program hits breakpoint at *`heap_trace_start()`*. Trace data will be saved to `/tmp/heap_log.svdat`. Tracing will be stopped when program hits breakpoint at *`heap_trace_stop()`*.

4. Run GDB using the following command `xtensa-esp32-elf-gdb -x gdbinit </path/to/program/elf>`
5. Quit GDB when program stops at *`heap_trace_stop()`*. Trace data are saved in `/tmp/heap.svdat`
6. Run processing script `$IDF_PATH/tools/esp_app_trace/sysviewtrace_proc.py -p -b </path/to/program/elf> /tmp/heap_log.svdat`

The output from the heap trace will look something like this:

```
Parse trace from '/tmp/heap.svdat'...
Stop parsing trace. (Timeout 0.000000 sec while reading 1 bytes!)
Process events from '['/tmp/heap.svdat']'...
[0.002244575] HEAP: Allocated 1 bytes @ 0x3ffaffd8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002258425] HEAP: Allocated 2 bytes @ 0x3ffaffe0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:48
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002563725] HEAP: Freed bytes @ 0x3ffaffe0 from task "free" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:31 (discriminator 9)
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002782950] HEAP: Freed bytes @ 0x3ffb40b8 from task "main" on core 0 by:
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590

[0.002798700] HEAP: Freed bytes @ 0x3ffb50bc from task "main" on core 0 by:
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590

[0.102436025] HEAP: Allocated 2 bytes @ 0x3ffaffe0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.102449800] HEAP: Allocated 4 bytes @ 0x3ffaffe8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:48
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.102666150] HEAP: Freed bytes @ 0x3ffaffe8 from task "free" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:31 (discriminator 9)
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.202436200] HEAP: Allocated 3 bytes @ 0x3ffaffe8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.202451725] HEAP: Allocated 6 bytes @ 0x3ffafff0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:48
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.202667075] HEAP: Freed bytes @ 0x3ffafff0 from task "free" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:31 (discriminator 9)
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.302436000] HEAP: Allocated 4 bytes @ 0x3ffafff0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.302451475] HEAP: Allocated 8 bytes @ 0x3ffb40b8 from task "alloc" on core 0 by:
```

(continues on next page)

(continued from previous page)

```
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:48
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.302667500] HEAP: Freed bytes @ 0x3ffb40b8 from task "free" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:31 (discriminator 9)
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

Processing completed.
Processed 1019 events
=============== HEAP TRACE REPORT ===============
Processed 14 heap events.
[0.002244575] HEAP: Allocated 1 bytes @ 0x3ffaffd8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.102436025] HEAP: Allocated 2 bytes @ 0x3ffaffe0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.202436200] HEAP: Allocated 3 bytes @ 0x3ffaffe8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.302436000] HEAP: Allocated 4 bytes @ 0x3ffafff0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

Found 10 leaked bytes in 4 blocks.
```

Heap Tracing To Find Heap Corruption Heap tracing can also be used to help track down heap corruption. When a region in heap is corrupted, it may be from some other part of the program which allocated memory at a nearby address.

If you have some idea at what time the corruption occurred, enabling heap tracing in `HEAP_TRACE_ALL` mode allows you to record all the functions which allocated memory, and the addresses of the allocations.

Using heap tracing in this way is very similar to memory leak detection as described above. For memory which is allocated and not freed, the output is the same. However, records will also be shown for memory which has been freed.

Performance Impact Enabling heap tracing in menuconfig increases the code size of your program, and has a very small negative impact on performance of heap allocation/free operations even when heap tracing is not running.

When heap tracing is running, heap allocation/free operations are substantially slower than when heap tracing is stopped. Increasing the depth of stack frames recorded for each allocation (see above) will also increase this performance impact.

False-Positive Memory Leaks Not everything printed by *`heap_trace_dump()`* is necessarily a memory leak. Among things which may show up here, but are not memory leaks:

- Any memory which is allocated after *`heap_trace_start()`* but then freed after *`heap_trace_stop()`* will appear in the leak dump.

- Allocations may be made by other tasks in the system. Depending on the timing of these tasks, it's quite possible this memory is freed after *heap_trace_stop()* is called.
- The first time a task uses stdio - for example, when it calls `printf()` - a lock (RTOS mutex semaphore) is allocated by the libc. This allocation lasts until the task is deleted.
- Certain uses of `printf()`, such as printing floating point numbers, will allocate some memory from the heap on demand. These allocations last until the task is deleted.
- The Bluetooth, Wi-Fi, and TCP/IP libraries will allocate heap memory buffers to handle incoming or outgoing data. These memory buffers are usually short-lived, but some may be shown in the heap leak trace if the data was received/transmitted by the lower levels of the network while the leak trace was running.
- TCP connections will continue to use some memory after they are closed, because of the `TIME_WAIT` state. After the `TIME_WAIT` period has completed, this memory will be freed.

One way to differentiate between "real" and "false positive" memory leaks is to call the suspect code multiple times while tracing is running, and look for patterns (multiple matching allocations) in the heap trace output.

API Reference - Heap Tracing

Header File

- components/heap/include/esp_heap_trace.h

Functions

esp_err_t **heap_trace_init_standalone** (*heap_trace_record_t* *record_buffer, size_t num_records)

Initialise heap tracing in standalone mode.

This function must be called before any other heap tracing functions.

To disable heap tracing and allow the buffer to be freed, stop tracing and then call heap_trace_init_standalone(NULL, 0);

Parameters
- **record_buffer** –Provide a buffer to use for heap trace data. Must remain valid any time heap tracing is enabled, meaning it must be allocated from internal memory not in PSRAM.
- **num_records** –Size of the heap trace buffer, as number of record structures.

Returns
- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing is currently in progress.
- ESP_OK Heap tracing initialised successfully.

esp_err_t **heap_trace_init_tohost** (void)

Initialise heap tracing in host-based mode.

This function must be called before any other heap tracing functions.

Returns
- ESP_ERR_INVALID_STATE Heap tracing is currently in progress.
- ESP_OK Heap tracing initialised successfully.

esp_err_t **heap_trace_start** (*heap_trace_mode_t* mode)

Start heap tracing. All heap allocations & frees will be traced, until heap_trace_stop() is called.

Note: heap_trace_init_standalone() must be called to provide a valid buffer, before this function is called.

Note: Calling this function while heap tracing is running will reset the heap trace state and continue tracing.

Parameters **mode** –Mode for tracing.

- HEAP_TRACE_ALL means all heap allocations and frees are traced.
- HEAP_TRACE_LEAKS means only suspected memory leaks are traced. (When memory is freed, the record is removed from the trace buffer.)

Returns

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE A non-zero-length buffer has not been set via heap_trace_init_standalone().
- ESP_OK Tracing is started.

esp_err_t **heap_trace_stop**(void)

Stop heap tracing.

Returns

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not in progress.
- ESP_OK Heap tracing stopped..

esp_err_t **heap_trace_resume**(void)

Resume heap tracing which was previously stopped.

Unlike heap_trace_start(), this function does not clear the buffer of any pre-existing trace records.

The heap trace mode is the same as when heap_trace_start() was last called (or HEAP_TRACE_ALL if heap_trace_start() was never called).

Returns

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was already started.
- ESP_OK Heap tracing resumed.

size_t **heap_trace_get_count**(void)

Return number of records in the heap trace buffer.

It is safe to call this function while heap tracing is running.

esp_err_t **heap_trace_get**(size_t index, *heap_trace_record_t* *record)

Return a raw record from the heap trace buffer.

Note: It is safe to call this function while heap tracing is running, however in HEAP_TRACE_LEAK mode record indexing may skip entries unless heap tracing is stopped first.

Parameters

- **index** –Index (zero-based) of the record to return.
- **record** –**[out]** Record where the heap trace record will be copied.

Returns

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not initialised.
- ESP_ERR_INVALID_ARG Index is out of bounds for current heap trace record count.
- ESP_OK Record returned successfully.

void **heap_trace_dump**(void)

Dump heap trace record data to stdout.

Note: It is safe to call this function while heap tracing is running, however in HEAP_TRACE_LEAK mode the dump may skip entries unless heap tracing is stopped first.

Structures

struct **heap_trace_record_t**

Trace record data type. Stores information about an allocated region of memory.

Public Members

uint32_t **ccount**

CCOUNT of the CPU when the allocation was made. LSB (bit value 1) is the CPU number (0 or 1).

void ***address**

Address which was allocated.

size_t **size**

Size of the allocation.

void ***alloced_by**[CONFIG_HEAP_TRACING_STACK_DEPTH]

Call stack of the caller which allocated the memory.

void ***freed_by**[CONFIG_HEAP_TRACING_STACK_DEPTH]

Call stack of the caller which freed the memory (all zero if not freed.)

Macros

CONFIG_HEAP_TRACING_STACK_DEPTH

Enumerations

enum **heap_trace_mode_t**

Values:

enumerator **HEAP_TRACE_ALL**

enumerator **HEAP_TRACE_LEAKS**

2.10.15 High Resolution Timer (ESP Timer)

Overview

Although FreeRTOS provides software timers, these timers have a few limitations:

- Maximum resolution is equal to RTOS tick period
- Timer callbacks are dispatched from a low-priority task

Hardware timers are free from both of the limitations, but often they are less convenient to use. For example, application components may need timer events to fire at certain times in the future, but the hardware timer only contains one "compare" value used for interrupt generation. This means that some facility needs to be built on top of the hardware timer to manage the list of pending events can dispatch the callbacks for these events as corresponding hardware interrupts happen.

An interrupt level of the handler depends on the *CONFIG_ESP_TIMER_INTERRUPT_LEVEL* option. It allows to set this: 1, 2 or 3 level (by default 1). Raising the level, the interrupt handler can reduce the timer processing delay.

`esp_timer` set of APIs provides one-shot and periodic timers, microsecond time resolution, and 64-bit range.

Internally, `esp_timer` uses a 64-bit hardware timer, where the implementation depends on the target. LAC timer is used for ESP32.

Timer callbacks can be dispatched by two methods:

- `ESP_TIMER_TASK`
- `ESP_TIMER_ISR`. Available only if *CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD* is enabled (by default disabled).

`ESP_TIMER_TASK`. Timer callbacks are dispatched from a high-priority `esp_timer` task. Because all the callbacks are dispatched from the same task, it is recommended to only do the minimal possible amount of work from the callback itself, posting an event to a lower priority task using a queue instead.

If other tasks with priority higher than `esp_timer` are running, callback dispatching will be delayed until `esp_timer` task has a chance to run. For example, this will happen if an SPI Flash operation is in progress.

`ESP_TIMER_ISR`. Timer callbacks are dispatched directly from the timer interrupt handler. This method is useful for some simple callbacks which aim for lower latency.

Creating and starting a timer, and dispatching the callback takes some time. Therefore, there is a lower limit to the timeout value of one-shot `esp_timer`. If *`esp_timer_start_once()`* is called with a timeout value less than 20us, the callback will be dispatched only after approximately 20us.

Periodic `esp_timer` also imposes a 50us restriction on the minimal timer period. Periodic software timers with period of less than 50us are not practical since they would consume most of the CPU time. Consider using dedicated hardware peripherals or DMA features if you find that a timer with small period is required.

Using `esp_timer` APIs

Single timer is represented by *`esp_timer_handle_t`* type. Timer has a callback function associated with it. This callback function is called from the `esp_timer` task each time the timer elapses.

- To create a timer, call *`esp_timer_create()`*.
- To delete the timer when it is no longer needed, call *`esp_timer_delete()`*.

The timer can be started in one-shot mode or in periodic mode.

- To start the timer in one-shot mode, call *`esp_timer_start_once()`*, passing the time interval after which the callback should be called. When the callback gets called, the timer is considered to be stopped.
- To start the timer in periodic mode, call *`esp_timer_start_periodic()`*, passing the period with which the callback should be called. The timer keeps running until *`esp_timer_stop()`* is called.

Note that the timer must not be running when *`esp_timer_start_once()`* or *`esp_timer_start_periodic()`* is called. To restart a running timer, call *`esp_timer_stop()`* first, then call one of the start functions.

Callback functions

Note: Keep the callback functions as short as possible otherwise it will affect all timers.

Timer callbacks which are processed by `ESP_TIMER_ISR` method should not call the context switch call - `portYIELD_FROM_ISR()`, instead of this you should use the *`esp_timer_isr_dispatch_need_yield()`* function. The context switch will be done after all ISR dispatch timers have been processed, if required by the system.

esp_timer during the light sleep

During light sleep, the esp_timer counter stops and no callback functions are called. Instead, the time is counted by the RTC counter. Upon waking up, the system gets the difference between the counters and calls a function that advances the esp_timer counter. Since the counter has been advanced, the system starts calling callbacks that were not called during sleep. The number of callbacks depends on the duration of the sleep and the period of the timers. It can lead to overflow of some queues. This only applies to periodic timers, one-shot timers will be called once.

This behavior can be changed by calling *`esp_timer_stop()`* before sleeping. In some cases, this can be inconvenient, and instead of the stop function, you can use the *skip_unhandled_events* option during *`esp_timer_create()`*. When the *skip_unhandled_events* is true, if a periodic timer expires one or more times during light sleep then only one callback is called on wake.

Using the *skip_unhandled_events* option with *automatic light sleep* (see *Power Management APIs*) helps to reduce the consumption of the system when it is in light sleep. The duration of light sleep is also determined by esp_timers. Timers with *skip_unhandled_events* option will not wake up the system.

Handling callbacks

esp_timer is designed to achieve a high-resolution low latency timer and the ability to handle delayed events. If the timer is late then the callback will be called as soon as possible, it will not be lost. In the worst case, when the timer has not been processed for more than one period (for periodic timers), in this case the callbacks will be called one after the other without waiting for the set period. This can be bad for some applications, and the *skip_unhandled_events* option was introduced to eliminate this behavior. If *skip_unhandled_events* is set then a periodic timer that has expired multiple times without being able to call the callback will still result in only one callback event once processing is possible.

Obtaining Current Time

`esp_timer` also provides a convenience function to obtain the time passed since start-up, with microsecond precision: *`esp_timer_get_time()`*. This function returns the number of microseconds since `esp_timer` was initialized, which usually happens shortly before `app_main` function is called.

Unlike *gettimeofday* function, values returned by *`esp_timer_get_time()`*:

- Start from zero after the chip wakes up from deep sleep
- Do not have timezone or DST adjustments applied

Application Example

The following example illustrates usage of `esp_timer` APIs: system/esp_timer.

API Reference

Header File

- components/esp_timer/include/esp_timer.h

Functions

esp_err_t **esp_timer_early_init** (void)

Minimal initialization of esp_timer.

This function can be called very early in startup process, after this call only esp_timer_get_time function can be used.

Note: This function is called from startup code. Applications do not need to call this function before using other esp_timer APIs.

Returns

- ESP_OK on success

esp_err_t **esp_timer_init** (void)

Initialize esp_timer library.

Note: This function is called from startup code. Applications do not need to call this function before using other esp_timer APIs. Before calling this function, esp_timer_early_init must be called by the startup code.

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_STATE if already initialized
- other errors from interrupt allocator

esp_err_t **esp_timer_deinit** (void)

De-initialize esp_timer library.

Note: Normally this function should not be called from applications

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not yet initialized

esp_err_t **esp_timer_create** (const *esp_timer_create_args_t* *create_args, *esp_timer_handle_t* *out_handle)

Create an esp_timer instance.

Note: When done using the timer, delete it with esp_timer_delete function.

Parameters

- **create_args** –Pointer to a structure with timer creation arguments. Not saved by the library, can be allocated on the stack.
- **out_handle** –**[out]** Output, pointer to esp_timer_handle_t variable which will hold the created timer handle.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if some of the create_args are not valid
- ESP_ERR_INVALID_STATE if esp_timer library is not initialized yet
- ESP_ERR_NO_MEM if memory allocation fails

esp_err_t **esp_timer_start_once**(*esp_timer_handle_t* timer, uint64_t timeout_us)

Start one-shot timer.

Timer should not be running when this function is called.

Parameters
- **timer** –timer handle created using esp_timer_create
- **timeout_us** –timer timeout, in microseconds relative to the current moment

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running

esp_err_t **esp_timer_start_periodic**(*esp_timer_handle_t* timer, uint64_t period)

Start a periodic timer.

Timer should not be running when this function is called. This function will start the timer which will trigger every 'period' microseconds.

Parameters
- **timer** –timer handle created using esp_timer_create
- **period** –timer period, in microseconds

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running

esp_err_t **esp_timer_restart**(*esp_timer_handle_t* timer, uint64_t timeout_us)

Restart a currently running timer.

If the given timer is a one-shot timer, the timer is restarted immediately and will timeout once in `timeout_us` microseconds. If the given timer is a periodic timer, the timer is restarted immediately with a new period of `timeout_us` microseconds.

Parameters
- **timer** –timer Handle created using esp_timer_create
- **timeout_us** –Timeout, in microseconds relative to the current time. In case of a periodic timer, also represents the new period.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is not running

esp_err_t **esp_timer_stop**(*esp_timer_handle_t* timer)

Stop the timer.

This function stops the timer previously started using esp_timer_start_once or esp_timer_start_periodic.

Parameters **timer** –timer handle created using esp_timer_create

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if the timer is not running

esp_err_t **esp_timer_delete**(*esp_timer_handle_t* timer)

Delete an esp_timer instance.

The timer must be stopped before deleting. A one-shot timer which has expired does not need to be stopped.

Parameters **timer** –timer handle allocated using esp_timer_create

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if the timer is running

int64_t **esp_timer_get_time**(void)

Get time in microseconds since boot.

Returns number of microseconds since underlying timer has been started

int64_t **esp_timer_get_next_alarm**(void)

Get the timestamp when the next timeout is expected to occur.

Returns Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by esp_timer_get_time.

int64_t **esp_timer_get_next_alarm_for_wake_up**(void)

Get the timestamp when the next timeout is expected to occur skipping those which have skip_unhandled_events flag.

Returns Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by esp_timer_get_time.

esp_err_t **esp_timer_get_period**(*esp_timer_handle_t* timer, uint64_t *period)

Get the period of a timer.

This function fetches the timeout period of a timer.

Note: The timeout period is the time interval with which a timer restarts after expiry. For one-shot timers, the period is 0 as there is no periodicity associated with such timers.

Parameters

- **timer** –timer handle allocated using esp_timer_create
- **period** –memory to store the timer period value in microseconds

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the arguments are invalid

esp_err_t **esp_timer_get_expiry_time**(*esp_timer_handle_t* timer, uint64_t *expiry)

Get the expiry time of a one-shot timer.

This function fetches the expiry time of a one-shot timer.

Note: This API returns a valid expiry time only for a one-shot timer. It returns an error if the timer handle passed to the function is for a periodic timer.

Parameters

- **timer** –timer handle allocated using esp_timer_create
- **expiry** –memory to store the timeout value in microseconds

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the arguments are invalid
- ESP_ERR_NOT_SUPPORTED if the timer type is periodic

esp_err_t **esp_timer_dump**(FILE *stream)

Dump the list of timers to a stream.

If CONFIG_ESP_TIMER_PROFILING option is enabled, this prints the list of all the existing timers. Otherwise, only the list active timers is printed.

The format is:

name period alarm times_armed times_triggered total_callback_run_time

where:

name —timer name (if CONFIG_ESP_TIMER_PROFILING is defined), or timer pointer period —period of timer, in microseconds, or 0 for one-shot timer alarm - time of the next alarm, in microseconds since boot, or 0 if the timer is not started

The following fields are printed if CONFIG_ESP_TIMER_PROFILING is defined:

times_armed —number of times the timer was armed via esp_timer_start_X times_triggered - number of times the callback was called total_callback_run_time - total time taken by callback to execute, across all calls

Parameters **`stream`** –stream (such as stdout) to dump the information to
Returns
- ESP_OK on success
- ESP_ERR_NO_MEM if can not allocate temporary buffer for the output

void **`esp_timer_isr_dispatch_need_yield`**(void)

Requests a context switch from a timer callback function.

This only works for a timer that has an ISR dispatch method. The context switch will be called after all ISR dispatch timers have been processed.

bool **`esp_timer_is_active`**(*esp_timer_handle_t* timer)

Returns status of a timer, active or not.

This function is used to identify if the timer is still active or not.

Parameters **`timer`** –timer handle created using esp_timer_create
Returns
- 1 if timer is still active
- 0 if timer is not active.

Structures

struct **`esp_timer_create_args_t`**

Timer configuration passed to esp_timer_create.

Public Members

esp_timer_cb_t **`callback`**

Function to call when timer expires.

void ***`arg`**

Argument to pass to the callback.

esp_timer_dispatch_t **`dispatch_method`**

Call the callback from task or from ISR.

const char ***`name`**

Timer name, used in esp_timer_dump function.

bool **`skip_unhandled_events`**

Skip unhandled events for periodic timers.

Type Definitions

typedef struct esp_timer ***`esp_timer_handle_t`**

Opaque type representing a single esp_timer.

typedef void (*`esp_timer_cb_t`)(void *arg)

Timer callback function type.

Param arg pointer to opaque user-specific data

Enumerations

enum `esp_timer_dispatch_t`

Method for dispatching timer callback.

Values:

enumerator `ESP_TIMER_TASK`

Callback is called from timer task.

enumerator `ESP_TIMER_MAX`

Count of the methods for dispatching timer callback.

2.10.16 Internal and Unstable APIs

This section is listing some APIs that are internal or likely to be changed or removed in the next releases of ESP-IDF.

API Reference

Header File

- components/esp_rom/include/esp_rom_sys.h

Functions

int `esp_rom_printf`(const char *fmt, ...)

Print formated string to console device.

Note: float and long long data are not supported!

Parameters

- `fmt` –Format string
- `...` –Additional arguments, depending on the format string

Returns int: Total number of characters written on success; A negative number on failure.

void `esp_rom_delay_us`(uint32_t us)

Pauses execution for us microseconds.

Parameters `us` –Number of microseconds to pause

void `esp_rom_install_channel_putc`(int channel, void (*putc)(char c))

esp_rom_printf can print message to different channels simultaneously. This function can help install the low level putc function for esp_rom_printf.

Parameters

- `channel` –Channel number (startting from 1)
- `putc` –Function pointer to the putc implementation. Set NULL can disconnect esp_rom_printf with putc.

void **esp_rom_install_uart_printf**(void)

Install UART1 as the default console channel, equivalent to `esp_rom_install_channel_putc(1, esp_rom_uart_putc)`

soc_reset_reason_t **esp_rom_get_reset_reason**(int cpu_no)

Get reset reason of CPU.

Parameters **cpu_no** –CPU number
Returns Reset reason code (see in soc/reset_reasons.h)

void **esp_rom_route_intr_matrix**(int cpu_core, uint32_t periph_intr_id, uint32_t cpu_intr_num)

Route peripheral interrupt sources to CPU's interrupt port by matrix.

Usually there're 4 steps to use an interrupt:

a. Route peripheral interrupt source to CPU. e.g. esp_rom_route_intr_matrix(0, ETS_WIFI_MAC_INTR_SOURCE, ETS_WMAC_INUM)
b. Set interrupt handler for CPU
c. Enable CPU interupt
d. Enable peripheral interrupt

Parameters
- **cpu_core** –The CPU number, which the peripheral interupt will inform to
- **periph_intr_id** –The peripheral interrupt source number
- **cpu_intr_num** –The CPU interrupt number

uint32_t **esp_rom_get_cpu_ticks_per_us**(void)

Get the real CPU ticks per us.

Returns CPU ticks per us

2.10.17 Inter-Processor Call

Note: The IPC is an **Inter-Processor Call** and **NOT Inter-Process Communication** as found on other operating systems.

Overview

Due to the dual core nature of the ESP32, there are instances where a certain callback must be run in the context of a particular CPU such as:

- When allocating an ISR to an interrupt source of a particular CPU (applies to freeing a particular CPU's interrupt source as well).
- On particular chips (such as the ESP32), accessing memory that is exclusive to a particular CPU (such as RTC Fast Memory).
- Reading the registers/state of another CPU.

The IPC (Inter-Processor Call) feature allows a particular CPU (the calling CPU) to trigger the execution of a callback function on another CPU (the target CPU). The IPC feature allows execution of a callback function on the target CPU in either a task context, or a High Priority Interrupt context (see *High-Level Interrupts* for more details). Depending on the context that the callback function is executed in, different restrictions apply to the implementation of the callback function.

IPC in Task Context

The IPC feature implements callback execution in a task context by creating an IPC task for each CPU during application startup. When the calling CPU needs to execute a callback on the target CPU, the callback will execute in the context of the target CPU's IPC task.

When using IPCs in a task context, users need to consider the following:

- IPC callbacks should ideally be simple and short. **An IPC callback should avoid attempting to block or yield**.
- The IPC tasks are created at the highest possible priority (i.e., `configMAX_PRIORITIES - 1`) thus the callback should also run at that priority as a result. However, *CONFIG_ESP_IPC_USES_CALLERS_PRIORITY* is enabled by default which will temporarily lower the priority of the target CPU's IPC task to the calling CPU before executing the callback.
- Depending on the complexity of the callback, users may need to configure the stack size of the IPC task via *CONFIG_ESP_IPC_TASK_STACK_SIZE*.
- The IPC feature is internally protected by a mutex. Therefore, simultaneous IPC calls from two or more calling CPUs will be handled on a first come first serve basis.

API Usage Task Context IPC callbacks have the following restrictions:

- The callback must be of type `void func(void *arg)`
- The callback should avoid attempting to block or yield as this will result in the target CPU's IPC task blocking or yielding.
- The callback must avoid changing any aspect of the IPC task (e.g., by calling `vTaskPrioritySet(NULL, x)`).

The IPC feature offers the API listed below to execute a callback in a task context on a target CPU. The API allows the calling CPU to block until the callback's execution has completed, or return immediately once the callback's execution has started.

- *`esp_ipc_call()`* will trigger an IPC call on the target CPU. This function will block until the target CPU's IPC task **begins** execution of the callback.
- *`esp_ipc_call_blocking()`* will trigger an IPC on the target CPU. This function will block until the target CPU's IPC task **completes** execution of the callback.

IPC in ISR Context

In some cases, we need to quickly obtain the state of another CPU such as in a core dump, GDB stub, various unit tests, and DPORT workaround. For such scenarios, the IPC feature supports execution of callbacks in a *High Priority Interrupt* context. The IPC feature implements the High Priority Interrupt context by reserving a High Priority Interrupt on each CPU for IPC usage. When a calling CPU needs to execute a callback on the target CPU, the callback will execute in the context of the High Priority Interrupt of the target CPU.

When using IPCs in High Priority Interrupt context, users need to consider the following:

- Since the callback is executed in a High Priority Interrupt context, the callback must be written entirely in assembly. See the API Usage below for more details regarding writing assembly callbacks.
- The priority of the reserved High Priority Interrupt is dependent on the *CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL* option
- **When the callback executes:**
  - The calling CPU will disable interrupts of level 3 and lower
  - Although the priority of the reserved interrupt depends on *CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL*, during the execution IPC ISR callback, the target CPU will disable interrupts of level 5 and lower regardless of what *CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL* is set to.

API Usage High Priority Interrupt IPC callbacks have the following restrictions:

- The callback must be of type `void func(void *arg)` but implemented entirely in assembly

- **The callback is invoked via the `CALLX0` instruction with register windowing disabled, thus the callback:**
  - Must not call any register window related instructions (e.g., `entry` and `retw`).
  - Must not call other C functions as register windowing is disabled
- The callback should be placed in IRAM at a 4-byte aligned address
- **(On invocation of/after returning from) the callback, the registers `a2, a3, a4` are (saved/restored) automatically thu**
  - `a2` will contain the `void *arg` of the callback
  - `a3/a4` are free to use as scratch registers

The IPC feature offers the API listed below to execute a callback in a High Priority Interrupt context.

- *`esp_ipc_isr_asm_call()`* will trigger an IPC call on the target CPU. This function will busy-wait until the target CPU begins execution of the callback.
- *`esp_ipc_isr_asm_call_blocking()`* will trigger an IPC call on the target CPU. This function will busy-wait until the target CPU completes execution of the callback.

The following code-blocks demonstrates a High Priority Interrupt IPC callback written in assembly that simply reads the target CPU's cycle count.

```
/* esp_test_ipc_isr_get_cycle_count_other_cpu(void *arg) */
// this function reads CCOUNT of the target CPU and stores it in arg.
// use only a2, a3 and a4 regs here.
.section    .iram1, "ax"
.align      4
.global     esp_test_ipc_isr_get_cycle_count_other_cpu
.type       esp_test_ipc_isr_get_cycle_count_other_cpu, @function
// Args:
// a2 - void* arg
esp_test_ipc_isr_get_cycle_count_other_cpu:
rsr.ccount a3
s32i    a3, a2, 0
ret
```

```
unit32_t cycle_count;
esp_ipc_isr_asm_call_blocking(esp_test_ipc_isr_get_cycle_count_other_cpu, (void␣
↪*)cycle_count);
```

Note: The number of scratch registers available for use is sufficient for most simple use cases. But if your callback requires more scratch registers, `void *arg` can point to a buffer that is used as a register save area. The callback can then save and restore more registers. See the system/ipc/ipc_isr.

Note: For more examples of High Priority Interrupt IPC callbacks, see components/esp_system/port/arch/xtensa/esp_ipc_isr_routines.S and :*components/esp_system/test/test_ipc_isr.S*

The High Priority Interrupt IPC API also provides the following convenience functions that can stall/resume the target CPU. These API utilize the High Priority Interrupt IPC, but supply their own internal callbacks:

- *`esp_ipc_isr_stall_other_cpu()`* stalls the target CPU. The calling CPU disables interrupts of level 3 and lower while the target CPU will busy-wait with interrupts of level 5 and lower disabled. The target CPU will busy-wait until *`esp_ipc_isr_release_other_cpu()`* is called.
- *`esp_ipc_isr_release_other_cpu()`* resumes the target CPU.

API Reference

Header File

- components/esp_system/include/esp_ipc.h

Functions

esp_err_t **esp_ipc_call**(uint32_t cpu_id, *esp_ipc_func_t* func, void *arg)

Execute a callback on a given CPU.

Execute a given callback on a particular CPU. The callback must be of type "esp_ipc_func_t" and will be invoked in the context of the target CPU's IPC task.

- This function will block the target CPU's IPC task has begun execution of the callback
- If another IPC call is ongoing, this function will block until the ongoing IPC call completes
- The stack size of the IPC task can be configured via the CONFIG_ESP_IPC_TASK_STACK_SIZE option

Note: In single-core mode, returns ESP_ERR_INVALID_ARG for cpu_id 1.

Parameters

- **cpu_id** –**[in]** CPU where the given function should be executed (0 or 1)
- **func** –**[in]** Pointer to a function of type void func(void* arg) to be executed
- **arg** –**[in]** Arbitrary argument of type void* to be passed into the function

Returns

- ESP_ERR_INVALID_ARG if cpu_id is invalid
- ESP_ERR_INVALID_STATE if the FreeRTOS scheduler is not running
- ESP_OK otherwise

esp_err_t **esp_ipc_call_blocking**(uint32_t cpu_id, *esp_ipc_func_t* func, void *arg)

Execute a callback on a given CPU until and block until it completes.

This function is identical to esp_ipc_call() except that this function will block until the execution of the callback completes.

Note: In single-core mode, returns ESP_ERR_INVALID_ARG for cpu_id 1.

Parameters

- **cpu_id** –**[in]** CPU where the given function should be executed (0 or 1)
- **func** –**[in]** Pointer to a function of type void func(void* arg) to be executed
- **arg** –**[in]** Arbitrary argument of type void* to be passed into the function

Returns

- ESP_ERR_INVALID_ARG if cpu_id is invalid
- ESP_ERR_INVALID_STATE if the FreeRTOS scheduler is not running
- ESP_OK otherwise

Type Definitions

typedef void (***esp_ipc_func_t**)(void *arg)

IPC Callback.

A callback of this type should be provided as an argument when calling esp_ipc_call() or esp_ipc_call_blocking().

Header File

- components/esp_system/include/esp_ipc_isr.h

Functions

void **esp_ipc_isr_asm_call**(*esp_ipc_isr_func_t* func, void *arg)

Execute an assembly callback on the other CPU.

Execute a given callback on the other CPU in the context of a High Priority Interrupt.

- This function will busy-wait in a critical section until the other CPU has started execution of the callback
- The callback must be written in assembly, is invoked using a CALLX0 instruction, and has a2, a3, a4 as scratch registers. See docs for more details

Note: This function is not available in single-core mode.

Parameters

- **func** –**[in]** Pointer to a function of type void func(void* arg) to be executed
- **arg** –**[in]** Arbitrary argument of type void* to be passed into the function

void **esp_ipc_isr_asm_call_blocking**(*esp_ipc_isr_func_t* func, void *arg)

Execute an assembly callback on the other CPU and busy-wait until it completes.

This function is identical to esp_ipc_isr_asm_call() except that this function will busy-wait until the execution of the callback completes.

Note: This function is not available in single-core mode.

Parameters

- **func** –**[in]** Pointer to a function of type void func(void* arg) to be executed
- **arg** –**[in]** Arbitrary argument of type void* to be passed into the function

void **esp_ipc_isr_stall_other_cpu**(void)

Stall the other CPU.

This function will stall the other CPU. The other CPU is stalled by busy-waiting in the context of a High Priority Interrupt. The other CPU will not be resumed until esp_ipc_isr_release_other_cpu() is called.

- This function is internally implemented using IPC ISR
- This function is used for DPORT workaround.
- If the stall feature is paused using esp_ipc_isr_stall_pause(), this function will have no effect

Note: This function is not available in single-core mode.

Note: It is the caller’s responsibility to avoid deadlocking on spinlocks

void **esp_ipc_isr_release_other_cpu**(void)

Release the other CPU.

This function will release the other CPU that was previously stalled from calling esp_ipc_isr_stall_other_cpu()

- This function is used for DPORT workaround.
- If the stall feature is paused using esp_ipc_isr_stall_pause(), this function will have no effect

Note: This function is not available in single-core mode.

void **esp_ipc_isr_stall_pause**(void)

Puase the CPU stall feature.

This function will pause the CPU stall feature. Once paused, calls to esp_ipc_isr_stall_other_cpu() and esp_ipc_isr_release_other_cpu() will have no effect. If a IPC ISR call is already in progress, this function will busy-wait until the call completes before pausing the CPU stall feature.

void **esp_ipc_isr_stall_abort**(void)

Abort a CPU stall.

This function will abort any stalling routine of the other CPU due to a pervious call to esp_ipc_isr_stall_other_cpu(). This function aborts the stall in a non-recoverable manner, thus should only be called in case of a panic().

- This function is used in panic handling code

void **esp_ipc_isr_stall_resume**(void)

Resume the CPU stall feature.

This function will resume the CPU stall feature that was previously paused by calling esp_ipc_isr_stall_pause(). Once resumed, calls to esp_ipc_isr_stall_other_cpu() and esp_ipc_isr_release_other_cpu() will have effect again.

Type Definitions

typedef void (***esp_ipc_isr_func_t**)(void *arg)

IPC ISR Callback.

A callback of this type should be provided as an argument when calling esp_ipc_isr_asm_call() or esp_ipc_isr_asm_call_blocking().

2.10.18 Interrupt allocation

Overview

The ESP32 has two cores, with 32 interrupts each. Each interrupt has a certain priority level, most (but not all) interrupts are connected to the interrupt mux.

Because there are more interrupt sources than interrupts, sometimes it makes sense to share an interrupt in multiple drivers. The *esp_intr_alloc()* abstraction exists to hide all these implementation details.

A driver can allocate an interrupt for a certain peripheral by calling *esp_intr_alloc()* (or *esp_intr_alloc_intrstatus()*). It can use the flags passed to this function to set the type of interrupt allocated, specifying a particular level or trigger method. The interrupt allocation code will then find an applicable interrupt, use the interrupt mux to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

This code presents two different types of interrupts, handled differently: shared interrupts and non-shared interrupts. The simplest ones are non-shared interrupts: a separate interrupt is allocated per *esp_intr_alloc()* call and this interrupt is solely used for the peripheral attached to it, with only one ISR that will get called. On the other hand, shared interrupts can have multiple peripherals triggering them, with multiple ISRs being called when one of the peripherals attached signals an interrupt. Thus, ISRs that are intended for shared interrupts should check the interrupt status of the peripheral they service in order to check if any action is required.

Non-shared interrupts can be either level- or edge-triggered. Shared interrupts can only be level interrupts due to the chance of missed interrupts when edge interrupts are used.

For example, let's say DevA and DevB share an interrupt. DevB signals an interrupt, so INT line goes high. The ISR handler calls code for DevA but does nothing. Then, ISR handler calls code for DevB, but while doing that, DevA signals an interrupt. DevB's ISR is done, it clears interrupt status for DevB and exits interrupt code. Now, an interrupt for DevA is still pending, but because the INT line never went low, as DevA kept it high even when the interrupt for DevB was cleared, the interrupt is never serviced.

Multicore issues

Peripherals that can generate interrupts can be divided in two types:

- External peripherals, within the ESP32 but outside the Xtensa cores themselves. Most ESP32 peripherals are of this type.
- Internal peripherals, part of the Xtensa CPU cores themselves.

Interrupt handling differs slightly between these two types of peripherals.

Internal peripheral interrupts Each Xtensa CPU core has its own set of six internal peripherals:

- Three timer comparators
- A performance monitor
- Two software interrupts.

Internal interrupt sources are defined in esp_intr_alloc.h as `ETS_INTERNAL_*_INTR_SOURCE`.

These peripherals can only be configured from the core they are associated with. When generating an interrupt, the interrupt they generate is hard-wired to their associated core; it's not possible to have, for example, an internal timer comparator of one core generate an interrupt on another core. That is why these sources can only be managed using a task running on that specific core. Internal interrupt sources are still allocatable using *`esp_intr_alloc()`* as normal, but they cannot be shared and will always have a fixed interrupt level (namely, the one associated in hardware with the peripheral).

External Peripheral Interrupts The remaining interrupt sources are from external peripherals. These are defined in soc/soc.h as `ETS_*_INTR_SOURCE`.

Non-internal interrupt slots in both CPU cores are wired to an interrupt multiplexer, which can be used to route any external interrupt source to any of these interrupt slots.

- Allocating an external interrupt will always allocate it on the core that does the allocation.
- Freeing an external interrupt must always happen on the same core it was allocated on.
- Disabling and enabling external interrupts from another core is allowed.
- Multiple external interrupt sources can share an interrupt slot by passing `ESP_INTR_FLAG_SHARED` as a flag to *`esp_intr_alloc()`*.

Care should be taken when calling *`esp_intr_alloc()`* from a task which is not pinned to a core. During task switching, these tasks can migrate between cores. Therefore it is impossible to tell which CPU the interrupt is allocated on, which makes it difficult to free the interrupt handle and may also cause debugging difficulties. It is advised to use *`xTaskCreatePinnedToCore()`* with a specific CoreID argument to create tasks that will allocate interrupts. In the case of internal interrupt sources, this is required.

IRAM-Safe Interrupt Handlers

The `ESP_INTR_FLAG_IRAM` flag registers an interrupt handler that always runs from IRAM (and reads all its data from DRAM), and therefore does not need to be disabled during flash erase and write operations.

This is useful for interrupts which need a guaranteed minimum execution latency, as flash write and erase operations can be slow (erases can take tens or hundreds of milliseconds to complete).

It can also be useful to keep an interrupt handler in IRAM if it is called very frequently, to avoid flash cache misses.

Refer to the *SPI flash API documentation* for more details.

Multiple Handlers Sharing A Source

Several handlers can be assigned to a same source, given that all handlers are allocated using the `ESP_INTR_FLAG_SHARED` flag. They will all be allocated to the interrupt, which the source is attached to, and called sequentially when the source is active. The handlers can be disabled and freed individually. The source is attached to the interrupt (enabled), if one or more handlers are enabled, otherwise detached. A handler will never be called when disabled, while **its source may still be triggered** if any one of its handler enabled.

Sources attached to non-shared interrupt do not support this feature.

Though the framework support this feature, you have to use it *very carefully*. There usually exist two ways to stop an interrupt from being triggered: *disable the source* or *mask peripheral interrupt status*. IDF only handles enabling and disabling of the source itself, leaving status and mask bits to be handled by users. **Status bits shall either be masked before the handler responsible for it is disabled, either be masked and then properly handled in another enabled interrupt**. Please note that leaving some status bits unhandled without masking them, while disabling the handlers for them, will cause the interrupt(s) to be triggered indefinitely, resulting therefore in a system crash.

API Reference

Header File

- components/esp_hw_support/include/esp_intr_alloc.h

Functions

esp_err_t **esp_intr_mark_shared**(int intno, int cpu, bool is_in_iram)

Mark an interrupt as a shared interrupt.

This will mark a certain interrupt on the specified CPU as an interrupt that can be used to hook shared interrupt handlers to.

Parameters

- **intno** –The number of the interrupt (0-31)
- **cpu** –CPU on which the interrupt should be marked as shared (0 or 1)
- **is_in_iram** –Shared interrupt is for handlers that reside in IRAM and the int can be left enabled while the flash cache is disabled.

Returns ESP_ERR_INVALID_ARG if cpu or intno is invalid ESP_OK otherwise

esp_err_t **esp_intr_reserve**(int intno, int cpu)

Reserve an interrupt to be used outside of this framework.

This will mark a certain interrupt on the specified CPU as reserved, not to be allocated for any reason.

Parameters

- **intno** –The number of the interrupt (0-31)
- **cpu** –CPU on which the interrupt should be marked as shared (0 or 1)

Returns ESP_ERR_INVALID_ARG if cpu or intno is invalid ESP_OK otherwise

esp_err_t **esp_intr_alloc**(int source, int flags, *intr_handler_t* handler, void *arg, *intr_handle_t* *ret_handle)

Allocate an interrupt with the given parameters.

This finds an interrupt that matches the restrictions as given in the flags parameter, maps the given interrupt source to it and hooks up the given interrupt handler (with optional argument) as well. If needed, it can return a handle for the interrupt as well.

The interrupt will always be allocated on the core that runs this function.

If ESP_INTR_FLAG_IRAM flag is used, and handler address is not in IRAM or RTC_FAST_MEM, then ESP_ERR_INVALID_ARG is returned.

Parameters

- **source** –The interrupt source. One of the ETS_*_INTR_SOURCE interrupt mux sources, as defined in soc/soc.h, or one of the internal ETS_INTERNAL_*_INTR_SOURCE sources as defined in this header.
- **flags** –An ORred mask of the ESP_INTR_FLAG_* defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- **handler** –The interrupt handler. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- **arg** –Optional argument for passed to the interrupt handler
- **ret_handle** –Pointer to an intr_handle_t to store a handle that can later be used to request details or free the interrupt. Can be NULL if no handle is required.

Returns ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_ERR_NOT_FOUND No free interrupt found with the specified flags ESP_OK otherwise

esp_err_t **esp_intr_alloc_intrstatus**(int source, int flags, uint32_t intrstatusreg, uint32_t intrstatusmask, *intr_handler_t* handler, void *arg, *intr_handle_t* *ret_handle)

Allocate an interrupt with the given parameters.

This essentially does the same as esp_intr_alloc, but allows specifying a register and mask combo. For shared interrupts, the handler is only called if a read from the specified register, ANDed with the mask, returns non-zero. By passing an interrupt status register address and a fitting mask, this can be used to accelerate interrupt handling in the case a shared interrupt is triggered; by checking the interrupt statuses first, the code can decide which ISRs can be skipped

Parameters

- **source** –The interrupt source. One of the ETS_*_INTR_SOURCE interrupt mux sources, as defined in soc/soc.h, or one of the internal ETS_INTERNAL_*_INTR_SOURCE sources as defined in this header.
- **flags** –An ORred mask of the ESP_INTR_FLAG_* defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- **intrstatusreg** –The address of an interrupt status register
- **intrstatusmask** –A mask. If a read of address intrstatusreg has any of the bits that are 1 in the mask set, the ISR will be called. If not, it will be skipped.
- **handler** –The interrupt handler. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- **arg** –Optional argument for passed to the interrupt handler
- **ret_handle** –Pointer to an intr_handle_t to store a handle that can later be used to request details or free the interrupt. Can be NULL if no handle is required.

Returns ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_ERR_NOT_FOUND No free interrupt found with the specified flags ESP_OK otherwise

esp_err_t **esp_intr_free**(*intr_handle_t* handle)

Disable and free an interrupt.

Use an interrupt handle to disable the interrupt and release the resources associated with it. If the current core is not the core that registered this interrupt, this routine will be assigned to the core that allocated this interrupt, blocking and waiting until the resource is successfully released.

Note: When the handler shares its source with other handlers, the interrupt status bits it's responsible for should be managed properly before freeing it. see `esp_intr_disable` for more details. Please do not call this function in `esp_ipc_call_blocking`.

Parameters **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
Returns ESP_ERR_INVALID_ARG the handle is NULL ESP_FAIL failed to release this handle ESP_OK otherwise

int **esp_intr_get_cpu**(*intr_handle_t* handle)

Get CPU number an interrupt is tied to.

Parameters **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
Returns The core number where the interrupt is allocated

int **esp_intr_get_intno**(*intr_handle_t* handle)

Get the allocated interrupt for a certain handle.

Parameters **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
Returns The interrupt number

esp_err_t **esp_intr_disable**(*intr_handle_t* handle)

Disable the interrupt associated with the handle.

Note:

a. For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.
b. When several handlers sharing a same interrupt source, interrupt status bits, which are handled in the handler to be disabled, should be masked before the disabling, or handled in other enabled interrupts properly. Miss of interrupt status handling will cause infinite interrupt calls and finally system crash.

Parameters **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
Returns ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

esp_err_t **esp_intr_enable**(*intr_handle_t* handle)

Enable the interrupt associated with the handle.

Note: For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.

Parameters **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
Returns ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

esp_err_t **esp_intr_set_in_iram**(*intr_handle_t* handle, bool is_in_iram)

Set the "in IRAM" status of the handler.

Note: Does not work on shared interrupts.

Parameters

- **handle** –The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
- **is_in_iram** –Whether the handler associated with this handle resides in IRAM. Handlers residing in IRAM can be called when cache is disabled.

Returns ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

void **esp_intr_noniram_disable**(void)

Disable interrupts that aren't specifically marked as running from IRAM.

void **esp_intr_noniram_enable**(void)

Re-enable interrupts disabled by esp_intr_noniram_disable.

void **esp_intr_enable_source**(int inum)

enable the interrupt source based on its number

Parameters **inum** –interrupt number from 0 to 31

void **esp_intr_disable_source**(int inum)

disable the interrupt source based on its number

Parameters **inum** –interrupt number from 0 to 31

static inline int **esp_intr_flags_to_level**(int flags)

Get the lowest interrupt level from the flags.

Parameters **flags** –The same flags that pass to `esp_intr_alloc_intrstatus` API

Macros

ESP_INTR_FLAG_LEVEL1

Interrupt allocation flags.

These flags can be used to specify which interrupt qualities the code calling esp_intr_alloc* needs. Accept a Level 1 interrupt vector (lowest priority)

ESP_INTR_FLAG_LEVEL2

Accept a Level 2 interrupt vector.

ESP_INTR_FLAG_LEVEL3

Accept a Level 3 interrupt vector.

ESP_INTR_FLAG_LEVEL4

Accept a Level 4 interrupt vector.

ESP_INTR_FLAG_LEVEL5

Accept a Level 5 interrupt vector.

ESP_INTR_FLAG_LEVEL6

Accept a Level 6 interrupt vector.

ESP_INTR_FLAG_NMI

Accept a Level 7 interrupt vector (highest priority)

ESP_INTR_FLAG_SHARED

Interrupt can be shared between ISRs.

ESP_INTR_FLAG_EDGE

Edge-triggered interrupt.

ESP_INTR_FLAG_IRAM

ISR can be called if cache is disabled.

ESP_INTR_FLAG_INTRDISABLED

Return with this interrupt disabled.

ESP_INTR_FLAG_LOWMED

Low and medium prio interrupts. These can be handled in C.

ESP_INTR_FLAG_HIGH

High level interrupts. Need to be handled in assembly.

ESP_INTR_FLAG_LEVELMASK

Mask for all level flags.

ETS_INTERNAL_TIMER0_INTR_SOURCE

Platform timer 0 interrupt source.

The esp_intr_alloc* functions can allocate an int for all ETS_*_INTR_SOURCE interrupt sources that are routed through the interrupt mux. Apart from these sources, each core also has some internal sources that do not pass through the interrupt mux. To allocate an interrupt for these sources, pass these pseudo-sources to the functions.

ETS_INTERNAL_TIMER1_INTR_SOURCE

Platform timer 1 interrupt source.

ETS_INTERNAL_TIMER2_INTR_SOURCE

Platform timer 2 interrupt source.

ETS_INTERNAL_SW0_INTR_SOURCE

Software int source 1.

ETS_INTERNAL_SW1_INTR_SOURCE

Software int source 2.

ETS_INTERNAL_PROFILING_INTR_SOURCE

Int source for profiling.

ETS_INTERNAL_UNUSED_INTR_SOURCE

Interrupt is not assigned to any source.

ETS_INTERNAL_INTR_SOURCE_OFF

Provides SystemView with positive IRQ IDs, otherwise scheduler events are not shown properly

ESP_INTR_ENABLE(inum)

Enable interrupt by interrupt number

ESP_INTR_DISABLE(inum)

Disable interrupt by interrupt number

Type Definitions

typedef void (***intr_handler_t**)(void *arg)

Function prototype for interrupt handler function

typedef struct *intr_handle_data_t* **intr_handle_data_t**

Interrupt handler associated data structure

typedef *intr_handle_data_t* ***intr_handle_t**

Handle to an interrupt handler

2.10.19 Logging library

Overview

The logging library provides two ways for setting log verbosity:

- **At compile time**: in menuconfig, set the verbosity level using the option *CONFIG_LOG_DEFAULT_LEVEL*.
- Optionally, also in menuconfig, set the maximum verbosity level using the option *CONFIG_LOG_MAXIMUM_LEVEL*. By default this is the same as the default level, but it can be set higher in order to compile more optional logs into the firmware.
- **At runtime**: all logs for verbosity levels lower than *CONFIG_LOG_DEFAULT_LEVEL* are enabled by default. The function *esp_log_level_set()* can be used to set a logging level on a per module basis. Modules are identified by their tags, which are human-readable ASCII zero-terminated strings.

There are the following verbosity levels:

- Error (lowest)
- Warning
- Info
- Debug
- Verbose (highest)

Note: The function *esp_log_level_set()* cannot set logging levels higher than specified by *CONFIG_LOG_MAXIMUM_LEVEL*. To increase log level for a specific file above this maximum at compile time, use the macro *LOG_LOCAL_LEVEL* (see the details below).

How to use this library

In each C file that uses logging functionality, define the TAG variable as shown below:

```
static const char* TAG = "MyModule";
```

Then use one of logging macros to produce output, e.g:

```
ESP_LOGW(TAG, "Baud rate error %.1f%%. Requested: %d baud, actual: %d baud", error␣
↪* 100, baud_req, baud_real);
```

Several macros are available for different verbosity levels:

- ESP_LOGE - error (lowest)
- ESP_LOGW - warning
- ESP_LOGI - info
- ESP_LOGD - debug
- ESP_LOGV - verbose (highest)

Additionally, there are ESP_EARLY_LOGx versions for each of these macros, e.g. *ESP_EARLY_LOGE*. These versions have to be used explicitly in the early startup code only, before heap allocator and syscalls have been initialized. Normal ESP_LOGx macros can also be used while compiling the bootloader, but they will fall back to the same implementation as ESP_EARLY_LOGx macros.

There are also ESP_DRAM_LOGx versions for each of these macros, e.g. *ESP_DRAM_LOGE*. These versions are used in some places where logging may occur with interrupts disabled or with flash cache inaccessible. Use of this macros should be as sparing as possible, as logging in these types of code should be avoided for performance reasons.

Note: Inside critical sections interrupts are disabled so it's only possible to use ESP_DRAM_LOGx (preferred) or ESP_EARLY_LOGx. Even though it's possible to log in these situations, it's better if your program can be structured not to require it.

To override default verbosity level at file or component scope, define the LOG_LOCAL_LEVEL macro.

At file scope, define it before including esp_log.h, e.g.:

```
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"
```

At component scope, define it in the component makefile:

```
target_compile_definitions(${COMPONENT_LIB} PUBLIC "-DLOG_LOCAL_LEVEL=ESP_LOG_
↪VERBOSE")
```

To configure logging output per module at runtime, add calls to the function *esp_log_level_set()* as follows:

```
esp_log_level_set("*", ESP_LOG_ERROR);        // set all components to ERROR level
esp_log_level_set("wifi", ESP_LOG_WARN);      // enable WARN logs from WiFi stack
esp_log_level_set("dhcpc", ESP_LOG_INFO);     // enable INFO logs from DHCP client
```

Note: The "DRAM" and "EARLY" log macro variants documented above do not support per module setting of log verbosity. These macros will always log at the "default" verbosity level, which can only be changed at runtime by calling esp_log_level("*", level).

Logging to Host via JTAG By default, the logging library uses the vprintf-like function to write formatted output to the dedicated UART. By calling a simple API, all log output may be routed to JTAG instead, making logging several times faster. For details, please refer to Section *Logging to Host*.

Application Example

The logging library is commonly used by most esp-idf components and examples. For demonstration of log functionality, check ESP-IDF's examples directory. The most relevant examples that deal with logging are the following:

- system/ota
- storage/sd_card
- protocols/https_request

API Reference

Header File

- components/log/include/esp_log.h

Functions

void **esp_log_level_set**(const char *tag, *esp_log_level_t* level)

Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

Note: Note that this function can not raise log level above the level set using CONFIG_LOG_MAXIMUM_LEVEL setting in menuconfig. To raise log level above the default one for a given file, define LOG_LOCAL_LEVEL to one of the ESP_LOG_* values, before including esp_log.h in this file.

Parameters

- **tag** –Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "*" resets log level for all tags to the given value.
- **level** –Selects log level to enable. Only logs at this and lower verbosity levels will be shown.

esp_log_level_t **esp_log_level_get**(const char *tag)

Get log level for a given tag, can be used to avoid expensive log statements.

Parameters **tag** –Tag of the log to query current level. Must be a non-NULL zero terminated string.

Returns The current log level for the given tag

vprintf_like_t **esp_log_set_vprintf**(*vprintf_like_t* func)

Set function used to output log entries.

By default, log output goes to UART0. This function can be used to redirect log output to some other destination, such as file or network. Returns the original log handler, which may be necessary to return output to the previous destination.

Note: Please note that function callback here must be re-entrant as it can be invoked in parallel from multiple thread context.

Parameters **func** –new Function used for output. Must have same signature as vprintf.

Returns func old Function used for output.

uint32_t **esp_log_timestamp**(void)

Function which returns timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros. In the 2nd stage bootloader, and at early application startup stage this function uses CPU cycle counter as time source. Later when FreeRTOS scheduler start running, it switches to FreeRTOS tick count.

For now, we ignore millisecond counter overflow.

Returns timestamp, in milliseconds

char ***esp_log_system_timestamp**(void)

Function which returns system timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros to print the system time as "HH:MM:SS.sss". The system time is initialized to 0 on startup, this can be set to the correct time with an SNTP sync, or manually with standard POSIX time functions.

Currently, this will not get used in logging from binary blobs (i.e. Wi-Fi & Bluetooth libraries), these will still print the RTOS tick time.

Returns timestamp, in "HH:MM:SS.sss"

uint32_t **esp_log_early_timestamp**(void)

Function which returns timestamp to be used in log output.

This function uses HW cycle counter and does not depend on OS, so it can be safely used after application crash.

Returns timestamp, in milliseconds

void **esp_log_write**(*esp_log_level_t* level, const char *tag, const char *format, ...)

Write message into the log.

This function is not intended to be used directly. Instead, use one of ESP_LOGE, ESP_LOGW, ESP_LOGI, ESP_LOGD, ESP_LOGV macros.

This function or these macros should not be used from an interrupt.

void **esp_log_writev**(*esp_log_level_t* level, const char *tag, const char *format, va_list args)

Write message into the log, va_list variant.

This function is provided to ease integration toward other logging framework, so that esp_log can be used as a log sink.

See also:

esp_log_write()

Macros

ESP_LOG_BUFFER_HEX_LEVEL(tag, buffer, buff_len, level)

Log a buffer of hex bytes at specified level, separated into 16 bytes each line.

Parameters

- **tag** –description tag
- **buffer** –Pointer to the buffer array
- **buff_len** –length of buffer in bytes
- **level** –level of the log

ESP_LOG_BUFFER_CHAR_LEVEL(tag, buffer, buff_len, level)

Log a buffer of characters at specified level, separated into 16 bytes each line. Buffer should contain only printable characters.

Parameters

- **tag** –description tag
- **buffer** –Pointer to the buffer array
- **buff_len** –length of buffer in bytes
- **level** –level of the log

ESP_LOG_BUFFER_HEXDUMP(tag, buffer, buff_len, level)

Dump a buffer to the log at specified level.

The dump log shows just like the one below:

```
 W (195) log_example: 0x3ffb4280   45 53 50 33 32 20 69 73  20 67 72 65 61 74␣
↪2c 20  |ESP32 is great, |
 W (195) log_example: 0x3ffb4290   77 6f 72 6b 69 6e 67 20  61 6c 6f 6e 67 20␣
↪77 69  |working along wi|
 W (205) log_example: 0x3ffb42a0   74 68 20 74 68 65 20 49  44 46 2e 00       ␣
↪       |th the IDF..|
```

It is highly recommended to use terminals with over 102 text width.

Parameters

- **tag** –description tag
- **buffer** –Pointer to the buffer array
- **buff_len** –length of buffer in bytes
- **level** –level of the log

ESP_LOG_BUFFER_HEX (tag, buffer, buff_len)

Log a buffer of hex bytes at Info level.

See also:

`esp_log_buffer_hex_level`

Parameters

- **tag** –description tag
- **buffer** –Pointer to the buffer array
- **buff_len** –length of buffer in bytes

ESP_LOG_BUFFER_CHAR (tag, buffer, buff_len)

Log a buffer of characters at Info level. Buffer should contain only printable characters.

See also:

`esp_log_buffer_char_level`

Parameters

- **tag** –description tag
- **buffer** –Pointer to the buffer array
- **buff_len** –length of buffer in bytes

ESP_EARLY_LOGE (tag, format, ...)

macro to output logs in startup code, before heap allocator and syscalls have been initialized. Log at `ESP_LOG_ERROR` level.

See also:

`printf`,`ESP_LOGE`,`ESP_DRAM_LOGE` In the future, we want to switch to C++20. We also want to become compatible with clang. Hence, we provide two versions of the following macros which are using variadic arguments. The first one is using the GNU extension ##__VA_ARGS__. The second one is using the C++20 feature **VA_OPT**(,). This allows users to compile their code with standard C++20 enabled instead of the GNU extension. Below C++20, we haven't found any good alternative to using ##__VA_ARGS__.

ESP_EARLY_LOGW (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_WARN` level.

See also:

`ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGI (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_INFO` level.

See also:

`ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGD (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_DEBUG` level.

See also:

`ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGV (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_VERBOSE` level.

See also:

`ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

_ESP_LOG_EARLY_ENABLED (log_level)

ESP_LOG_EARLY_IMPL (tag, format, log_level, log_tag_letter, ...)

ESP_LOGE (tag, format, ...)

ESP_LOGW (tag, format, ...)

ESP_LOGI (tag, format, ...)

ESP_LOGD (tag, format, ...)

ESP_LOGV (tag, format, ...)

ESP_LOG_LEVEL (level, tag, format, ...)

runtime macro to output logs at a specified level.

See also:

`printf`

Parameters

- **tag** –tag of the log, which can be used to change the log level by `esp_log_level_set` at runtime.
- **level** –level of the output log.
- **format** –format of the output log. See `printf`
- **...** –variables to be replaced into the log. See `printf`

ESP_LOG_LEVEL_LOCAL (level, tag, format, ...)

runtime macro to output logs at a specified level. Also check the level with `LOG_LOCAL_LEVEL`.

See also:

`printf`, `ESP_LOG_LEVEL`

ESP_DRAM_LOGE (tag, format, ...)

Macro to output logs when the cache is disabled. Log at `ESP_LOG_ERROR` level.

Similar to

Usage: `ESP_DRAM_LOGE(DRAM_STR("my_tag"), "format",` or`ESP_DRAM_LOGE(TAG,` “format” , ⋯)`, where TAG is a char* that points to a str in the DRAM.

See also:

`ESP_EARLY_LOGE`, the log level cannot be changed per-tag, however esp_log_level_set(“*” , level) will set the default level which controls these log lines also.

See also:

`esp_rom_printf`,`ESP_LOGE`

Note: Unlike normal logging macros, it ' s possible to use this macro when interrupts are disabled or inside an ISR.

Note: Placing log strings in DRAM reduces available DRAM, so only use when absolutely essential.

ESP_DRAM_LOGW(tag, format, ...)

macro to output logs when the cache is disabled at `ESP_LOG_WARN` level.

See also:

`ESP_DRAM_LOGW`,`ESP_LOGW`, `esp_rom_printf`

ESP_DRAM_LOGI(tag, format, ...)

macro to output logs when the cache is disabled at `ESP_LOG_INFO` level.

See also:

`ESP_DRAM_LOGI`,`ESP_LOGI`, `esp_rom_printf`

ESP_DRAM_LOGD(tag, format, ...)

macro to output logs when the cache is disabled at `ESP_LOG_DEBUG` level.

See also:

`ESP_DRAM_LOGD`,`ESP_LOGD`, `esp_rom_printf`

ESP_DRAM_LOGV(tag, format, ...)

macro to output logs when the cache is disabled at `ESP_LOG_VERBOSE` level.

See also:

`ESP_DRAM_LOGV`,`ESP_LOGV`, `esp_rom_printf`

Type Definitions

typedef int (***vprintf_like_t**)(const char*, va_list)

Enumerations

enum **esp_log_level_t**

Log level.

Values:

enumerator **ESP_LOG_NONE**

No log output

enumerator **ESP_LOG_ERROR**

Critical errors, software module can not recover on its own

enumerator **ESP_LOG_WARN**

Error conditions from which recovery measures have been taken

enumerator **ESP_LOG_INFO**

Information messages which describe normal flow of events

enumerator **ESP_LOG_DEBUG**

Extra information which is not necessary for normal use (values, pointers, sizes, etc).

enumerator **ESP_LOG_VERBOSE**

Bigger chunks of debugging information, or frequent messages which can potentially flood the output.

2.10.20 Miscellaneous System APIs

Software Reset

To perform software reset of the chip, the *esp_restart()* function is provided. When the function is called, execution of the program stops, both CPUs are reset, and the application is loaded by the bootloader and starts execution again.

Additionally, the *esp_register_shutdown_handler()* function can register a routine that will be automatically called before a restart (that is triggered by *esp_restart()*) occurs. This is similar to the functionality of `atexit` POSIX function.

Reset Reason

ESP-IDF applications can be started or restarted due to a variety of reasons. To get the last reset reason, call *esp_reset_reason()* function. See description of *esp_reset_reason_t* for the list of possible reset reasons.

Heap Memory

Two heap-memory-related functions are provided:

- *esp_get_free_heap_size()* returns the current size of free heap memory.
- *esp_get_minimum_free_heap_size()* returns the minimum size of free heap memory that has ever been available (i.e., the smallest size of free heap memory in the applications lifetime).

Note that ESP-IDF supports multiple heaps with different capabilities. The functions mentioned in this section return the size of heap memory that can be allocated using the `malloc` family of functions. For further information about heap memory, see *Heap Memory Allocation*.

MAC Address

These APIs allow querying and customizing MAC addresses for different supported network interfaces (e.g., Wi-Fi, Bluetooth, Ethernet).

To fetch the MAC address for a specific network interface (e.g., Wi-Fi, Bluetooth, Ethernet), call the function *esp_read_mac()*.

In ESP-IDF, the MAC addresses for the various network interfaces are calculated from a single *base MAC address*. By default, the Espressif base MAC address is used. This base MAC address is pre-programmed into the ESP32 eFuse in the factory during production.

Interface	MAC Address (4 universally administered, default)	MAC Address (2 universally administered)
Wi-Fi Station	base_mac	base_mac
Wi-Fi SoftAP	base_mac, +1 to the last octet	*Local MAC* (derived from Wi-Fi Station MAC)
Bluetooth	base_mac, +2 to the last octet	base_mac, +1 to the last octet
Ethernet	base_mac, +3 to the last octet	*Local MAC* (derived from Bluetooth MAC)

Note: The *configuration* configures the number of universally administered MAC addresses that are provided by Espressif.

Custom Base MAC The default base MAC is pre-programmed by Espressif in eFuse BLK0. To set a custom base MAC instead, call the function `esp_base_mac_addr_set()` before initializing any network interfaces or calling the `esp_read_mac()` function. The custom MAC address can be stored in any supported storage device (e.g., flash, NVS).

The custom base MAC addresses should be allocated such that derived MAC addresses will not overlap. Based on the table above, users can configure the option *CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES* to set the number of valid universal MAC addresses that can be derived from the custom base MAC.

Note: It is also possible to call the function `esp_netif_set_mac()` to set the specific MAC used by a network interface after network initialization. But it is recommended to use the base MAC approach documented here to avoid the possibility of the original MAC address briefly appearing on the network before being changed.

Custom MAC Address in eFuse When reading custom MAC addresses from eFuse, ESP-IDF provides a helper function `esp_efuse_mac_get_custom()`. This loads the MAC address from eFuse BLK3. This function assumes that the custom base MAC address is stored in the following format:

Field	# of bits	Range of bits	Notes
Version	8	191:184	0: invalid, others —valid
Reserved	128	183:56	
MAC address	48	55:8	
MAC address CRC	8	7:0	CRC-8-CCITT, polynomial 0x07

Note: If the 3/4 coding scheme is enabled, all eFuse fields in this block must be burnt at the same time.

Once MAC address has been obtained using `esp_efuse_mac_get_custom()`, call `esp_base_mac_addr_set()` to set this MAC address as base MAC address.

Local vs Universal MAC Addresses ESP32 comes pre-programmed with enough valid Espressif universally administered MAC addresses for all internal interfaces. The table above shows how to calculate and derive the MAC address for a specific interface according to the base MAC address.

When using a custom MAC address scheme, it is possible that not all interfaces can be assigned with a universally administered MAC address. In these cases, a locally administered MAC address is assigned. Note that these addresses are intended for use on a single local network only.

See this article for the definition of locally and universally administered MAC addresses.

Function *esp_derive_local_mac()* is called internally to derive a local MAC address from a universal MAC address. The process is as follows:

1. The U/L bit (bit value 0x2) is set in the first octet of the universal MAC address, creating a local MAC address.
2. If this bit is already set in the supplied universal MAC address (i.e., the supplied "universal" MAC address was in fact already a local MAC address), then the first octet of the local MAC address is XORed with 0x4.

Chip Version

esp_chip_info() function fills *esp_chip_info_t* structure with information about the chip. This includes the chip revision, number of CPU cores, and a bit mask of features enabled in the chip.

SDK Version

esp_get_idf_version() returns a string describing the ESP-IDF version which is used to compile the application. This is the same value as the one available through `IDF_VER` variable of the build system. The version string generally has the format of `git describe` output.

To get the version at build time, additional version macros are provided. They can be used to enable or disable parts of the program depending on the ESP-IDF version.

- *ESP_IDF_VERSION_MAJOR*, *ESP_IDF_VERSION_MINOR*, *ESP_IDF_VERSION_PATCH* are defined to integers representing major, minor, and patch version.
- *ESP_IDF_VERSION_VAL* and *ESP_IDF_VERSION* can be used when implementing version checks:

```
#include "esp_idf_version.h"

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)
    // enable functionality present in ESP-IDF v4.0
#endif
```

App Version

The application version is stored in *esp_app_desc_t* structure. It is located in DROM sector and has a fixed offset from the beginning of the binary file. The structure is located after *esp_image_header_t* and *esp_image_segment_header_t* structures. The type of the field version is string and it has a maximum length of 32 chars.

To set the version in your project manually, you need to set the `PROJECT_VER` variable in the `CMakeLists.txt` of your project. In application `CMakeLists.txt`, put `set(PROJECT_VER "0.1.0.1")` before including `project.cmake`.

If the *CONFIG_APP_PROJECT_VER_FROM_CONFIG* option is set, the value of *CONFIG_APP_PROJECT_VER* will be used. Otherwise, if the `PROJECT_VER` variable is not set in the project, it will be retrieved either from the `$(PROJECT_PATH)/version.txt` file (if present) or using git command `git describe`. If neither is available, `PROJECT_VER` will be set to "1". Application can make use of this by calling *esp_app_get_description()* or *esp_ota_get_partition_description()* functions.

API Reference

Header File

- components/esp_system/include/esp_system.h

Functions

esp_err_t **esp_register_shutdown_handler** (*shutdown_handler_t* handle)

Register shutdown handler.

This function allows you to register a handler that gets invoked before the application is restarted using esp_restart function.

Parameters **handle** –function to execute on restart

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if the handler has already been registered
- ESP_ERR_NO_MEM if no more shutdown handler slots are available

esp_err_t **esp_unregister_shutdown_handler** (*shutdown_handler_t* handle)

Unregister shutdown handler.

This function allows you to unregister a handler which was previously registered using esp_register_shutdown_handler function.

- ESP_OK on success
- ESP_ERR_INVALID_STATE if the given handler hasn't been registered before

void **esp_restart** (void)

Restart PRO and APP CPUs.

This function can be called both from PRO and APP CPUs. After successful restart, CPU reset reason will be SW_CPU_RESET. Peripherals (except for Wi-Fi, BT, UART0, SPI1, and legacy timers) are not reset. This function does not return.

esp_reset_reason_t **esp_reset_reason** (void)

Get reason of last reset.

Returns See description of esp_reset_reason_t for explanation of each value.

uint32_t **esp_get_free_heap_size** (void)

Get the size of available heap.

Note: Note that the returned value may be larger than the maximum contiguous block which can be allocated.

Returns Available heap size, in bytes.

uint32_t **esp_get_free_internal_heap_size** (void)

Get the size of available internal heap.

Note: Note that the returned value may be larger than the maximum contiguous block which can be allocated.

Returns Available internal heap size, in bytes.

uint32_t **esp_get_minimum_free_heap_size** (void)

Get the minimum heap that has ever been available.

Returns Minimum free heap ever available

void **esp_system_abort** (const char *details)

Trigger a software abort.

Parameters **details** –Details that will be displayed during panic handling.

Type Definitions

typedef void (*`shutdown_handler_t`)(void)

Shutdown handler type

Enumerations

enum `esp_reset_reason_t`

Reset reasons.

Values:

enumerator `ESP_RST_UNKNOWN`

Reset reason can not be determined.

enumerator `ESP_RST_POWERON`

Reset due to power-on event.

enumerator `ESP_RST_EXT`

Reset by external pin (not applicable for ESP32)

enumerator `ESP_RST_SW`

Software reset via esp_restart.

enumerator `ESP_RST_PANIC`

Software reset due to exception/panic.

enumerator `ESP_RST_INT_WDT`

Reset (software or hardware) due to interrupt watchdog.

enumerator `ESP_RST_TASK_WDT`

Reset due to task watchdog.

enumerator `ESP_RST_WDT`

Reset due to other watchdogs.

enumerator `ESP_RST_DEEPSLEEP`

Reset after exiting deep sleep mode.

enumerator `ESP_RST_BROWNOUT`

Brownout reset (software or hardware)

enumerator `ESP_RST_SDIO`

Reset over SDIO.

Header File

- components/esp_common/include/esp_idf_version.h

Functions

const char ***esp_get_idf_version**(void)

Return full IDF version string, same as 'git describe' output.

Note: If you are printing the ESP-IDF version in a log file or other information, this function provides more information than using the numerical version macros. For example, numerical version macros don't differentiate between development, pre-release and release versions, but the output of this function does.

Returns constant string from IDF_VER

Macros

ESP_IDF_VERSION_MAJOR

Major version number (X.x.x)

ESP_IDF_VERSION_MINOR

Minor version number (x.X.x)

ESP_IDF_VERSION_PATCH

Patch version number (x.x.X)

ESP_IDF_VERSION_VAL(major, minor, patch)

Macro to convert IDF version number into an integer

To be used in comparisons, such as ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)

ESP_IDF_VERSION

Current IDF version, as an integer

To be used in comparisons, such as ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)

Header File

- components/esp_hw_support/include/esp_mac.h

Functions

esp_err_t **esp_base_mac_addr_set**(const uint8_t *mac)

Set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage e.g. flash and EEPROM.

Base MAC address is used to generate the MAC addresses used by network interfaces.

If using a custom base MAC address, call this API before initializing any network interfaces. Refer to the ESP-IDF Programming Guide for details about how the Base MAC is used.

Note: Base MAC must be a unicast MAC (least significant bit of first byte must be zero).

Note: If not using a valid OUI, set the "locally administered" bit (bit value 0x02 in the first byte) to avoid collisions.

Parameters **mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)

Returns ESP_OK on success ESP_ERR_INVALID_ARG If mac is NULL or is not a unicast MAC

esp_err_t **esp_base_mac_addr_get**(uint8_t *mac)

Return base MAC address which is set using esp_base_mac_addr_set.

Note: If no custom Base MAC has been set, this returns the pre-programmed Espressif base MAC address.

Parameters **mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)

Returns ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL ESP_ERR_INVALID_MAC base MAC address has not been set

esp_err_t **esp_efuse_mac_get_custom**(uint8_t *mac)

Return base MAC address which was previously written to BLK3 of EFUSE.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. This API returns the custom base MAC address which was previously written to EFUSE BLK3 in a specified format.

Writing this EFUSE allows setting of a different (non-Espressif) base MAC address. It is also possible to store a custom base MAC address elsewhere, see esp_base_mac_addr_set() for details.

Note: This function is currently only supported on ESP32.

Parameters **mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)

Returns ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL ESP_ERR_INVALID_MAC CUSTOM_MAC address has not been set, all zeros (for esp32-xx) ESP_ERR_INVALID_VERSION An invalid MAC version field was read from BLK3 of EFUSE (for esp32) ESP_ERR_INVALID_CRC An invalid MAC CRC was read from BLK3 of EFUSE (for esp32)

esp_err_t **esp_efuse_mac_get_default**(uint8_t *mac)

Return base MAC address which is factory-programmed by Espressif in EFUSE.

Parameters **mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)

Returns ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL

esp_err_t **esp_read_mac**(uint8_t *mac, *esp_mac_type_t* type)

Read base MAC address and set MAC address of the interface.

This function first get base MAC address using esp_base_mac_addr_get(). Then calculates the MAC address of the specific interface requested, refer to ESP-IDF Programming Guide for the algorithm.

Parameters

- **mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)
- **type** –Type of MAC address to return

Returns ESP_OK on success

esp_err_t **esp_derive_local_mac**(uint8_t *local_mac, const uint8_t *universal_mac)

Derive local MAC address from universal MAC address.

This function copies a universal MAC address and then sets the "locally

administered" bit (bit 0x2) in the first octet, creating a locally administered MAC address.

If the universal MAC address argument is already a locally administered MAC address, then the first octet is XORed with 0x4 in order to create a different locally administered MAC address.

Parameters

- **local_mac** –base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64(used for IEEE 802.15.4)
- **universal_mac** –Source universal MAC address, length: 6 bytes.

Returns ESP_OK on success

Macros

MAC2STR(a)

MACSTR

Enumerations

enum **esp_mac_type_t**

Values:

enumerator **ESP_MAC_WIFI_STA**

enumerator **ESP_MAC_WIFI_SOFTAP**

enumerator **ESP_MAC_BT**

enumerator **ESP_MAC_ETH**

enumerator **ESP_MAC_IEEE802154**

Header File

- components/esp_hw_support/include/esp_chip_info.h

Functions

void **esp_chip_info**(*esp_chip_info_t* *out_info)

Fill an *esp_chip_info_t* structure with information about the chip.

Parameters **out_info** –**[out]** structure to be filled

Structures

struct **esp_chip_info_t**

The structure represents information about the chip.

Public Members

esp_chip_model_t **model**

chip model, one of esp_chip_model_t

uint32_t **features**

bit mask of CHIP_FEATURE_x feature flags

uint16_t **revision**

chip revision number (in format MXX; where M - wafer major version, XX - wafer minor version)

uint8_t **cores**

number of CPU cores

Macros

CHIP_FEATURE_EMB_FLASH

Chip has embedded flash memory.

CHIP_FEATURE_WIFI_BGN

Chip has 2.4GHz WiFi.

CHIP_FEATURE_BLE

Chip has Bluetooth LE.

CHIP_FEATURE_BT

Chip has Bluetooth Classic.

CHIP_FEATURE_IEEE802154

Chip has IEEE 802.15.4.

CHIP_FEATURE_EMB_PSRAM

Chip has embedded psram.

Enumerations

enum **esp_chip_model_t**

Chip models.

Values:

enumerator **CHIP_ESP32**

ESP32.

enumerator **CHIP_ESP32S2**

ESP32-S2.

enumerator **CHIP_ESP32S3**

ESP32-S3.

enumerator **CHIP_ESP32C3**

ESP32-C3.

enumerator **CHIP_ESP32H2**

ESP32-H2.

enumerator **CHIP_ESP32C2**

ESP32-C2.

Header File

- components/esp_hw_support/include/esp_cpu.h

Functions

void **esp_cpu_stall**(int core_id)

Stall a CPU core.

Parameters **core_id** –The core's ID

void **esp_cpu_unstall**(int core_id)

Resume a previously stalled CPU core.

Parameters **core_id** –The core's ID

void **esp_cpu_reset**(int core_id)

Reset a CPU core.

Parameters **core_id** –The core's ID

void **esp_cpu_wait_for_intr**(void)

Wait for Interrupt.

This function causes the current CPU core to execute its Wait For Interrupt (WFI or equivalent) instruction. After executing this function, the CPU core will stop execution until an interrupt occurs.

int **esp_cpu_get_core_id**(void)

Get the current core's ID.

This function will return the ID of the current CPU (i.e., the CPU that calls this function).

Returns The current core's ID [0..SOC_CPU_CORES_NUM - 1]

void ***esp_cpu_get_sp**(void)

Read the current stack pointer address.

Returns Stack pointer address

esp_cpu_cycle_count_t **esp_cpu_get_cycle_count**(void)

Get the current CPU core's cycle count.

Each CPU core maintains an internal counter (i.e., cycle count) that increments every CPU clock cycle.

Returns Current CPU's cycle count, 0 if not supported.

void **esp_cpu_set_cycle_count**(*esp_cpu_cycle_count_t* cycle_count)

Set the current CPU core's cycle count.

Set the given value into the internal counter that increments every CPU clock cycle.

Parameters **cycle_count** –CPU cycle count

void ***esp_cpu_pc_to_addr**(uint32_t pc)

Convert a program counter (PC) value to address.

If the architecture does not store the true virtual address in the CPU's PC or return addresses, this function will convert the PC value to a virtual address. Otherwise, the PC is just returned

Parameters **pc** –PC value
Returns Virtual address

void **esp_cpu_intr_get_desc**(int core_id, int intr_num, *esp_cpu_intr_desc_t* *intr_desc_ret)

Get a CPU interrupt's descriptor.

Each CPU interrupt has a descriptor describing the interrupt's capabilities and restrictions. This function gets the descriptor of a particular interrupt on a particular CPU.

Parameters
- **core_id** –**[in]** The core's ID
- **intr_num** –**[in]** Interrupt number
- **intr_desc_ret** –**[out]** The interrupt's descriptor

void **esp_cpu_intr_set_ivt_addr**(const void *ivt_addr)

Set the base address of the current CPU's Interrupt Vector Table (IVT)

Parameters **ivt_addr** –Interrupt Vector Table's base address

bool **esp_cpu_intr_has_handler**(int intr_num)

Check if a particular interrupt already has a handler function.

Check if a particular interrupt on the current CPU already has a handler function assigned.

Note: This function simply checks if the IVT of the current CPU already has a handler assigned.

Parameters **intr_num** –Interrupt number (from 0 to 31)
Returns True if the interrupt has a handler function, false otherwise.

void **esp_cpu_intr_set_handler**(int intr_num, *esp_cpu_intr_handler_t* handler, void *handler_arg)

Set the handler function of a particular interrupt.

Assign a handler function (i.e., ISR) to a particular interrupt on the current CPU.

Note: This function simply sets the handler function (in the IVT) and does not actually enable the interrupt.

Parameters
- **intr_num** –Interrupt number (from 0 to 31)
- **handler** –Handler function
- **handler_arg** –Argument passed to the handler function

void ***esp_cpu_intr_get_handler_arg**(int intr_num)

Get a handler function's argument of.

Get the argument of a previously assigned handler function on the current CPU.

Parameters **intr_num** –Interrupt number (from 0 to 31)
Returns The the argument passed to the handler function

void **esp_cpu_intr_enable**(uint32_t intr_mask)

Enable particular interrupts on the current CPU.

Parameters **intr_mask** –Bit mask of the interrupts to enable

void **esp_cpu_intr_disable**(uint32_t intr_mask)

Disable particular interrupts on the current CPU.

Parameters **intr_mask** –Bit mask of the interrupts to disable

uint32_t **esp_cpu_intr_get_enabled_mask**(void)

Get the enabled interrupts on the current CPU.

Returns Bit mask of the enabled interrupts

void **esp_cpu_intr_edge_ack**(int intr_num)

Acknowledge an edge interrupt.

Parameters **intr_num** –Interrupt number (from 0 to 31)

void **esp_cpu_configure_region_protection**(void)

Configure the CPU to disable access to invalid memory regions.

esp_err_t **esp_cpu_set_breakpoint**(int bp_num, const void *bp_addr)

Set and enable a hardware breakpoint on the current CPU.

Note: This function is meant to be called by the panic handler to set a breakpoint for an attached debugger during a panic.

Note: Overwrites previously set breakpoint with same breakpoint number.

Parameters

- **bp_num** –Hardware breakpoint number [0..SOC_CPU_BREAKPOINTS_NUM - 1]
- **bp_addr** –Address to set a breakpoint on

Returns ESP_OK if breakpoint is set. Failure otherwise

esp_err_t **esp_cpu_clear_breakpoint**(int bp_num)

Clear a hardware breakpoint on the current CPU.

Note: Clears a breakpoint regardless of whether it was previously set

---

Parameters **bp_num** –Hardware breakpoint number [0..SOC_CPU_BREAKPOINTS_NUM - 1]

Returns ESP_OK if breakpoint is cleared. Failure otherwise

esp_err_t **esp_cpu_set_watchpoint**(int wp_num, const void *wp_addr, size_t size, *esp_cpu_watchpoint_trigger_t* trigger)

Set and enable a hardware watchpoint on the current CPU.

Set and enable a hardware watchpoint on the current CPU, specifying the memory range and trigger operation. Watchpoints will break/panic the CPU when the CPU accesses (according to the trigger type) on a certain memory range.

Note: Overwrites previously set watchpoint with same watchpoint number. On RISC-V chips, this API uses method0(Exact matching) and method1(NAPOT matching) according to the riscv-debug-spec-0.13 specification for address matching. If the watch region size is 1byte, it uses exact matching (method 0). If the watch region size is larger than 1byte, it uses NAPOT matching (method 1). This mode requires the watching region start address to be aligned to the watching region size.

Parameters

- **wp_num** –Hardware watchpoint number [0..SOC_CPU_WATCHPOINTS_NUM - 1]
- **wp_addr** –Watchpoint' s base address, must be naturally aligned to the size of the region
- **size** –Size of the region to watch. Must be one of 2^n and in the range of [1 ··· SOC_CPU_WATCHPOINT_MAX_REGION_SIZE]
- **trigger** –Trigger type

Returns ESP_ERR_INVALID_ARG on invalid arg, ESP_OK otherwise

esp_err_t **esp_cpu_clear_watchpoint**(int wp_num)

Clear a hardware watchpoint on the current CPU.

Note: Clears a watchpoint regardless of whether it was previously set

Parameters **wp_num** – Hardware watchpoint number [0..SOC_CPU_WATCHPOINTS_NUM - 1]

Returns ESP_OK if watchpoint was cleared. Failure otherwise.

bool **esp_cpu_dbgr_is_attached**(void)

Check if the current CPU has a debugger attached.

Returns True if debugger is attached, false otherwise

void **esp_cpu_dbgr_break**(void)

Trigger a call to the current CPU's attached debugger.

intptr_t **esp_cpu_get_call_addr**(intptr_t return_address)

Given the return address, calculate the address of the preceding call instruction This is typically used to answer the question "where was the function called from?".

Parameters **return_address** – The value of the return address register. Typically set to the value of __builtin_return_address(0).

Returns Address of the call instruction preceding the return address.

bool **esp_cpu_compare_and_set**(volatile uint32_t *addr, uint32_t compare_value, uint32_t new_value)

Atomic compare-and-set operation.

Parameters

- **addr** – Address of atomic variable
- **compare_value** – Value to compare the atomic variable to
- **new_value** – New value to set the atomic variable to

Returns Whether the atomic variable was set or not

Structures

struct **esp_cpu_intr_desc_t**

CPU interrupt descriptor.

Each particular CPU interrupt has an associated descriptor describing that particular interrupt's characteristics. Call esp_cpu_intr_get_desc() to get the descriptors of a particular interrupt.

Public Members

int **priority**

Priority of the interrupt if it has a fixed priority, (-1) if the priority is configurable.

esp_cpu_intr_type_t **type**

Whether the interrupt is an edge or level type interrupt, ESP_CPU_INTR_TYPE_NA if the type is configurable.

uint32_t **flags**

Flags indicating extra details.

Macros

ESP_CPU_INTR_DESC_FLAG_SPECIAL

Interrupt descriptor flags of *esp_cpu_intr_desc_t*.

The interrupt is a special interrupt (e.g., a CPU timer interrupt)

ESP_CPU_INTR_DESC_FLAG_RESVD

The interrupt is reserved for internal use

Type Definitions

typedef uint32_t **esp_cpu_cycle_count_t**

CPU cycle count type.

This data type represents the CPU' s clock cycle count

typedef void (***esp_cpu_intr_handler_t**)(void *arg)

CPU interrupt handler type.

Enumerations

enum **esp_cpu_intr_type_t**

CPU interrupt type.

Values:

enumerator **ESP_CPU_INTR_TYPE_LEVEL**

enumerator **ESP_CPU_INTR_TYPE_EDGE**

enumerator **ESP_CPU_INTR_TYPE_NA**

enum **esp_cpu_watchpoint_trigger_t**

CPU watchpoint trigger type.

Values:

enumerator **ESP_CPU_WATCHPOINT_LOAD**

enumerator **ESP_CPU_WATCHPOINT_STORE**

enumerator **ESP_CPU_WATCHPOINT_ACCESS**

Header File

- components/esp_app_format/include/esp_app_desc.h

Functions

const *esp_app_desc_t* ***esp_app_get_description**(void)

Return esp_app_desc structure. This structure includes app version.

Return description for running app.

Returns Pointer to esp_app_desc structure.

int **esp_app_get_elf_sha256**(char *dst, size_t size)

Fill the provided buffer with SHA256 of the ELF file, formatted as hexadecimal, null-terminated. If the buffer size is not sufficient to fit the entire SHA256 in hex plus a null terminator, the largest possible number of bytes will be written followed by a null.

Parameters

- **dst** –Destination buffer
- **size** –Size of the buffer

Returns Number of bytes written to dst (including null terminator)

Structures

struct **esp_app_desc_t**

Description about application.

Public Members

uint32_t **magic_word**

Magic word ESP_APP_DESC_MAGIC_WORD

uint32_t **secure_version**

Secure version

uint32_t **reserv1**[2]

reserv1

char **version**[32]

Application version

char **project_name**[32]

Project name

char **time**[16]

Compile time

char **date**[16]

Compile date

char **idf_ver**[32]

Version IDF

uint8_t **app_elf_sha256**[32]

sha256 of elf file

uint32_t **reserv2**[20]

reserv2

Macros

ESP_APP_DESC_MAGIC_WORD

The magic word for the esp_app_desc structure that is in DROM.

2.10.21 Over The Air Updates (OTA)

OTA Process Overview

The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over Wi-Fi or Bluetooth.)

OTA requires configuring the *Partition Table* of the device with at least two "OTA app slot" partitions (i.e. *ota_0* and *ota_1*) and an "OTA Data Partition".

The OTA operation functions write a new app firmware image to whichever OTA app slot that is currently not selected for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

OTA Data Partition

An OTA data partition (type `data`, subtype `ota`) must be included in the *Partition Table* of any project which uses the OTA functions.

For factory boot settings, the OTA data partition should contain no data (all bytes erased to 0xFF). In this case the esp-idf software bootloader will boot the factory app if it is present in the partition table. If no factory app is included in the partition table, the first available OTA slot (usually `ota_0`) is booted.

After the first OTA update, the OTA data partition is updated to specify which OTA app slot partition should be booted next.

The OTA data partition is two flash sectors (0x2000 bytes) in size, to prevent problems if there is a power failure while it is being written. Sectors are independently erased and written with matching data, and if they disagree a counter field is used to determine which sector was written more recently.

App rollback

The main purpose of the application rollback is to keep the device working after the update. This feature allows you to roll back to the previous working application in case a new application has critical errors. When the rollback process is enabled and an OTA update provides a new version of the app, one of three things can happen:

- The application works fine, *`esp_ota_mark_app_valid_cancel_rollback()`* marks the running application with the state `ESP_OTA_IMG_VALID`. There are no restrictions on booting this application.
- The application has critical errors and further work is not possible, a rollback to the previous application is required, *`esp_ota_mark_app_invalid_rollback_and_reboot()`* marks the running application with the state `ESP_OTA_IMG_INVALID` and reset. This application will not be selected by the bootloader for boot and will boot the previously working application.
- If the *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is set, and a reset occurs without calling either function then the application is rolled back.

Note: The state is not written to the binary image of the application but rather to the `otadata` partition. The partition contains a `ota_seq` counter which is a pointer to the slot (ota_0, ota_1, ···) from which the application will be selected for boot.

App OTA State States control the process of selecting a boot app:

States	Restriction of selecting a boot app in bootloader
ESP_OTA_IMG_VALID	No restriction. Will be selected.
ESP_OTA_IMG_UNDEFINED	No restriction. Will be selected.
ESP_OTA_IMG_INVALID	Will not be selected.
ESP_OTA_IMG_ABORTED	Will not be selected.
ESP_OTA_IMG_NEW	If *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is set it will be selected only once. In bootloader the state immediately changes to ESP_OTA_IMG_PENDING_VERIFY.
ESP_OTA_IMG_PENDING_VERIFY	If *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is set it will not be selected, and the state will change to ESP_OTA_IMG_ABORTED.

If *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is not enabled (by default), then the use of the following functions *esp_ota_mark_app_valid_cancel_rollback()* and *esp_ota_mark_app_invalid_rollback_and_reboot()* are optional, and ESP_OTA_IMG_NEW and ESP_OTA_IMG_PENDING_VERIFY states are not used.

An option in Kconfig *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* allows you to track the first boot of a new application. In this case, the application must confirm its operability by calling *esp_ota_mark_app_valid_cancel_rollback()* function, otherwise the application will be rolled back upon reboot. It allows you to control the operability of the application during the boot phase. Thus, a new application has only one attempt to boot successfully.

Rollback Process The description of the rollback process when *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is enabled:

- The new application is successfully downloaded and *esp_ota_set_boot_partition()* function makes this partition bootable and sets the state ESP_OTA_IMG_NEW. This state means that the application is new and should be monitored for its first boot.
- Reboot *esp_restart()*.
- The bootloader checks for the ESP_OTA_IMG_PENDING_VERIFY state if it is set, then it will be written to ESP_OTA_IMG_ABORTED.
- The bootloader selects a new application to boot so that the state is not set as ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED.
- The bootloader checks the selected application for ESP_OTA_IMG_NEW state if it is set, then it will be written to ESP_OTA_IMG_PENDING_VERIFY. This state means that the application requires confirmation of its operability, if this does not happen and a reboot occurs, this state will be overwritten to ESP_OTA_IMG_ABORTED (see above) and this application will no longer be able to start, i.e. there will be a rollback to the previous working application.
- A new application has started and should make a self-test.
- If the self-test has completed successfully, then you must call the function *esp_ota_mark_app_valid_cancel_rollback()* because the application is awaiting confirmation of operability (ESP_OTA_IMG_PENDING_VERIFY state).
- If the self-test fails then call *esp_ota_mark_app_invalid_rollback_and_reboot()* function to roll back to the previous working application, while the invalid application is set ESP_OTA_IMG_INVALID state.
- If the application has not been confirmed, the state remains ESP_OTA_IMG_PENDING_VERIFY, and the next boot it will be changed to ESP_OTA_IMG_ABORTED. That will prevent re-boot of this application. There will be a rollback to the previous working application.

Unexpected Reset If a power loss or an unexpected crash occurs at the time of the first boot of a new application, it will roll back the application.

Recommendation: Perform the self-test procedure as quickly as possible, to prevent rollback due to power loss.

Only OTA partitions can be rolled back. Factory partition is not rolled back.

Booting invalid/aborted apps Booting an application which was previously set to ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED is possible:

- Get the last invalid application partition *esp_ota_get_last_invalid_partition()*.
- Pass the received partition to *esp_ota_set_boot_partition()*, this will update the otadata.
- Restart *esp_restart()*. The bootloader will boot the specified application.

To determine if self-tests should be run during startup of an application, call the *esp_ota_get_state_partition()* function. If result is ESP_OTA_IMG_PENDING_VERIFY then self-testing and subsequent confirmation of operability is required.

Where the states are set A brief description of where the states are set:

- ESP_OTA_IMG_VALID state is set by *esp_ota_mark_app_valid_cancel_rollback()* function.
- ESP_OTA_IMG_UNDEFINED state is set by *esp_ota_set_boot_partition()* function if *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is not enabled.
- ESP_OTA_IMG_NEW state is set by *esp_ota_set_boot_partition()* function if *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is enabled.
- ESP_OTA_IMG_INVALID state is set by *esp_ota_mark_app_invalid_rollback_and_reboot()* function.
- ESP_OTA_IMG_ABORTED state is set if there was no confirmation of the application operability and occurs reboots (if *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is enabled).
- ESP_OTA_IMG_PENDING_VERIFY state is set in a bootloader if *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* option is enabled and selected app has ESP_OTA_IMG_NEW state.

Anti-rollback

Anti-rollback prevents rollback to application with security version lower than one programmed in eFuse of chip.

This function works if set *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK* option. In the bootloader, when selecting a bootable application, an additional security version check is added which is on the chip and in the application image. The version in the bootable firmware must be greater than or equal to the version in the chip.

CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK and *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* options are used together. In this case, rollback is possible only on the security version which is equal or higher than the version in the chip.

A typical anti-rollback scheme is

- New firmware released with the elimination of vulnerabilities with the previous version of security.
- After the developer makes sure that this firmware is working. He can increase the security version and release a new firmware.
- Download new application.
- To make it bootable, run the function *esp_ota_set_boot_partition()*. If the security version of the new application is smaller than the version in the chip, the new application will be erased. Update to new firmware is not possible.
- Reboot.
- In the bootloader, an application with a security version greater than or equal to the version in the chip will be selected. If otadata is in the initial state, and one firmware was loaded via a serial channel, whose secure version is higher than the chip, then the secure version of efuse will be immediately updated in the bootloader.
- New application booted. Then the application should perform diagnostics of the operation and if it is completed successfully, you should call *esp_ota_mark_app_valid_cancel_rollback()* function to mark the running application with the ESP_OTA_IMG_VALID state and update the secure version on chip. Note that if was called *esp_ota_mark_app_invalid_rollback_and_reboot()* function a rollback may not happen as the device may not have any bootable apps. It will then return ESP_ERR_OTA_ROLLBACK_FAILED error and stay in the ESP_OTA_IMG_PENDING_VERIFY state.
- The next update of app is possible if a running app is in the ESP_OTA_IMG_VALID state.

Recommendation:

If you want to avoid the download/erase overhead in case of the app from the server has security version lower than the running app, you have to get `new_app_info.secure_version` from the first package of an image and compare it with the secure version of efuse. Use `esp_efuse_check_secure_version(new_app_info.secure_version)` function if it is true then continue downloading otherwise abort.

```
....
bool image_header_was_checked = false;
while (1) {
    int data_read = esp_http_client_read(client, ota_write_data, BUFFSIZE);
    ...
    if (data_read > 0) {
        if (image_header_was_checked == false) {
            esp_app_desc_t new_app_info;
            if (data_read > sizeof(esp_image_header_t) + sizeof(esp_image_segment_
↪header_t) + sizeof(esp_app_desc_t)) {
                // check current version with downloading
                if (esp_efuse_check_secure_version(new_app_info.secure_version) ==␣
↪false) {
                    ESP_LOGE(TAG, "This a new app can not be downloaded due to a␣
↪secure version is lower than stored in efuse.");
                    http_cleanup(client);
                    task_fatal_error();
                }

                image_header_was_checked = true;

                esp_ota_begin(update_partition, OTA_SIZE_UNKNOWN, &update_handle);
            }
        }
        esp_ota_write( update_handle, (const void *)ota_write_data, data_read);
    }
}
...
```

Restrictions:

- The number of bits in the `secure_version` field is limited to 32 bits. This means that only 32 times you can do an anti-rollback. You can reduce the length of this efuse field using *CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD* option.
- Anti-rollback works only if the encoding scheme for efuse is set to `NONE`.
- Factory and Test partitions are not supported in anti rollback scheme and hence partition table should not have partition with SubType set to `factory` or `test`.

`security_version`:

- In application image it is stored in `esp_app_desc` structure. The number is set *CONFIG_BOOTLOADER_APP_SECURE_VERSION*.
- In ESP32 it is stored in efuse `EFUSE_BLK3_RDATA4_REG`. (when a eFuse bit is programmed to 1, it can never be reverted to 0). The number of bits set in this register is the `security_version` from app.

Secure OTA Updates Without Secure boot

The verification of signed OTA updates can be performed even without enabling hardware secure boot. This can be achieved by setting *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT* and *CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT*

For more information refer to *Signed App Verification Without Hardware Secure Boot*

OTA Tool (otatool.py)

The component *app_update* provides a tool otatool.py for performing OTA partition-related operations on a target device. The following operations can be performed using the tool:

- read contents of otadata partition (read_otadata)
- erase otadata partition, effectively resetting device to factory app (erase_otadata)
- switch OTA partitions (switch_ota_partition)
- erasing OTA partition (erase_ota_partition)
- write to OTA partition (write_ota_partition)
- read contents of OTA partition (read_ota_partition)

The tool can either be imported and used from another Python script or invoked from shell script for users wanting to perform operation programmatically. This is facilitated by the tool's Python API and command-line interface, respectively.

Python API Before anything else, make sure that the *otatool* module is imported.

```
import sys
import os

idf_path = os.environ["IDF_PATH"]  # get value of IDF_PATH from environment
otatool_dir = os.path.join(idf_path, "components", "app_update")  # otatool.py␣
↪lives in $IDF_PATH/components/app_update

sys.path.append(otatool_dir)  # this enables Python to find otatool module
from otatool import *  # import all names inside otatool module
```

The starting point for using the tool's Python API to do is create a *OtatoolTarget* object:

```
# Create a partool.py target device connected on serial port /dev/ttyUSB1
target = OtatoolTarget("/dev/ttyUSB1")
```

The created object can now be used to perform operations on the target device:

```
# Erase otadata, reseting the device to factory app
target.erase_otadata()

# Erase contents of OTA app slot 0
target.erase_ota_partition(0)

# Switch boot partition to that of app slot 1
target.switch_ota_partition(1)

# Read OTA partition 'ota_3' and save contents to a file named 'ota_3.bin'
target.read_ota_partition("ota_3", "ota_3.bin")
```

The OTA partition to operate on is specified using either the app slot number or the partition name.

More information on the Python API is available in the docstrings for the tool.

Command-line Interface The command-line interface of *otatool.py* has the following structure:

```
otatool.py [command-args] [subcommand] [subcommand-args]

- command-args - these are arguments that are needed for executing the main␣
↪command (parttool.py), mostly pertaining to the target device
- subcommand - this is the operation to be performed
- subcommand-args - these are arguments that are specific to the chosen operation
```

```
# Erase otadata, resetting the device to factory app
otatool.py --port "/dev/ttyUSB1" erase_otadata

# Erase contents of OTA app slot 0
otatool.py --port "/dev/ttyUSB1" erase_ota_partition --slot 0

# Switch boot partition to that of app slot 1
otatool.py --port "/dev/ttyUSB1" switch_ota_partition --slot 1

# Read OTA partition 'ota_3' and save contents to a file named 'ota_3.bin'
otatool.py --port "/dev/ttyUSB1" read_ota_partition --name=ota_3 --output=ota_3.bin
```

More information can be obtained by specifying *–help* as argument:

```
# Display possible subcommands and show main command argument descriptions
otatool.py --help

# Show descriptions for specific subcommand arguments
otatool.py [subcommand] --help
```

See also

- *Partition Table documentation*
- *Lower-Level SPI Flash/Partition API*
- *ESP HTTPS OTA*

Application Example

End-to-end example of OTA firmware update workflow: system/ota.

API Reference

Header File

- components/app_update/include/esp_ota_ops.h

Functions

const *esp_app_desc_t* ***esp_ota_get_app_description**(void)

Return esp_app_desc structure. This structure includes app version.

Return description for running app.

Note: This API is present for backward compatibility reasons. Alternative function with the same functionality is `esp_app_get_description`

Returns Pointer to esp_app_desc structure.

int **esp_ota_get_app_elf_sha256**(char *dst, size_t size)

Fill the provided buffer with SHA256 of the ELF file, formatted as hexadecimal, null-terminated. If the buffer size is not sufficient to fit the entire SHA256 in hex plus a null terminator, the largest possible number of bytes will be written followed by a null.

Note: This API is present for backward compatibility reasons. Alternative function with the same functionality is `esp_app_get_elf_sha256`

Parameters
- **dst** –Destination buffer
- **size** –Size of the buffer

Returns Number of bytes written to dst (including null terminator)

esp_err_t **esp_ota_begin**(const *esp_partition_t* *partition, size_t image_size, *esp_ota_handle_t* *out_handle)

Commence an OTA update writing to the specified partition.

The specified partition is erased to the specified image size.

If image size is not yet known, pass OTA_SIZE_UNKNOWN which will cause the entire partition to be erased.

On success, this function allocates memory that remains in use until esp_ota_end() is called with the returned handle.

Note: If the rollback option is enabled and the running application has the ESP_OTA_IMG_PENDING_VERIFY state then it will lead to the ESP_ERR_OTA_ROLLBACK_INVALID_STATE error. Confirm the running app before to run download a new app, use esp_ota_mark_app_valid_cancel_rollback() function for it (this should be done as early as possible when you first download a new application).

Parameters
- **partition** –Pointer to info for partition which will receive the OTA update. Required.
- **image_size** –Size of new OTA app image. Partition will be erased in order to receive this size of image. If 0 or OTA_SIZE_UNKNOWN, the entire partition is erased.
- **out_handle** –On success, returns a handle which should be used for subsequent esp_ota_write() and esp_ota_end() calls.

Returns
- ESP_OK: OTA operation commenced successfully.
- ESP_ERR_INVALID_ARG: partition or out_handle arguments were NULL, or partition doesn't point to an OTA app partition.
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_OTA_PARTITION_CONFLICT: Partition holds the currently running firmware, cannot update in place.
- ESP_ERR_NOT_FOUND: Partition argument not found in partition table.
- ESP_ERR_OTA_SELECT_INFO_INVALID: The OTA data partition contains invalid data.
- ESP_ERR_INVALID_SIZE: Partition doesn't fit in configured flash size.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- ESP_ERR_OTA_ROLLBACK_INVALID_STATE: If the running app has not confirmed state. Before performing an update, the application must be valid.

esp_err_t **esp_ota_write**(*esp_ota_handle_t* handle, const void *data, size_t size)

Write OTA update data to partition.

This function can be called multiple times as data is received during the OTA operation. Data is written sequentially to the partition.

Parameters
- **handle** –Handle obtained from esp_ota_begin
- **data** –Data buffer to write
- **size** –Size of data buffer in bytes.

Returns
- ESP_OK: Data was written to flash successfully, or size = 0
- ESP_ERR_INVALID_ARG: handle is invalid.

- ESP_ERR_OTA_VALIDATE_FAILED: First byte of image contains invalid app image magic byte.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- ESP_ERR_OTA_SELECT_INFO_INVALID: OTA data partition has invalid contents

esp_err_t **esp_ota_write_with_offset**(*esp_ota_handle_t* handle, const void *data, size_t size, uint32_t offset)

Write OTA update data to partition at an offset.

This function can write data in non-contiguous manner. If flash encryption is enabled, data should be 16 bytes aligned.

Note: While performing OTA, if the packets arrive out of order, esp_ota_write_with_offset() can be used to write data in non-contiguous manner. Use of esp_ota_write_with_offset() in combination with esp_ota_write() is not recommended.

Parameters

- **handle** –Handle obtained from esp_ota_begin
- **data** –Data buffer to write
- **size** –Size of data buffer in bytes
- **offset** –Offset in flash partition

Returns

- ESP_OK: Data was written to flash successfully.
- ESP_ERR_INVALID_ARG: handle is invalid.
- ESP_ERR_OTA_VALIDATE_FAILED: First byte of image contains invalid app image magic byte.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- ESP_ERR_OTA_SELECT_INFO_INVALID: OTA data partition has invalid contents

esp_err_t **esp_ota_end**(*esp_ota_handle_t* handle)

Finish OTA update and validate newly written app image.

Note: After calling esp_ota_end(), the handle is no longer valid and any memory associated with it is freed (regardless of result).

Parameters **handle** –Handle obtained from esp_ota_begin().

Returns

- ESP_OK: Newly written OTA app image is valid.
- ESP_ERR_NOT_FOUND: OTA handle was not found.
- ESP_ERR_INVALID_ARG: Handle was never written to.
- ESP_ERR_OTA_VALIDATE_FAILED: OTA image is invalid (either not a valid app image, or - if secure boot is enabled - signature failed to verify.)
- ESP_ERR_INVALID_STATE: If flash encryption is enabled, this result indicates an internal error writing the final encrypted bytes to flash.

esp_err_t **esp_ota_abort**(*esp_ota_handle_t* handle)

Abort OTA update, free the handle and memory associated with it.

Parameters **handle** –obtained from esp_ota_begin().

Returns

- ESP_OK: Handle and its associated memory is freed successfully.
- ESP_ERR_NOT_FOUND: OTA handle was not found.

esp_err_t **esp_ota_set_boot_partition**(const *esp_partition_t* *partition)

Configure OTA data for a new boot partition.

Note: If this function returns ESP_OK, calling esp_restart() will boot the newly configured app partition.

Parameters **partition** –Pointer to info for partition containing app image to boot.

Returns

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_ERR_INVALID_ARG: partition argument was NULL or didn’t point to a valid OTA partition of type “app”.
- ESP_ERR_OTA_VALIDATE_FAILED: Partition contained invalid app image. Also returned if secure boot is enabled and signature validation failed.
- ESP_ERR_NOT_FOUND: OTA data partition not found.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash erase or write failed.

const *esp_partition_t* ***esp_ota_get_boot_partition**(void)

Get partition info of currently configured boot app.

If esp_ota_set_boot_partition() has been called, the partition which was set by that function will be returned.

If esp_ota_set_boot_partition() has not been called, the result is usually the same as esp_ota_get_running_partition(). The two results are not equal if the configured boot partition does not contain a valid app (meaning that the running partition will be an app that the bootloader chose via fallback).

If the OTA data partition is not present or not valid then the result is the first app partition found in the partition table. In priority order, this means: the factory app, the first OTA app slot, or the test app partition.

Note that there is no guarantee the returned partition is a valid app. Use esp_image_verify(ESP_IMAGE_VERIFY, ···) to verify if the returned partition contains a bootable image.

Returns Pointer to info for partition structure, or NULL if partition table is invalid or a flash read operation failed. Any returned pointer is valid for the lifetime of the application.

const *esp_partition_t* ***esp_ota_get_running_partition**(void)

Get partition info of currently running app.

This function is different to esp_ota_get_boot_partition() in that it ignores any change of selected boot partition caused by esp_ota_set_boot_partition(). Only the app whose code is currently running will have its partition information returned.

The partition returned by this function may also differ from esp_ota_get_boot_partition() if the configured boot partition is somehow invalid, and the bootloader fell back to a different app partition at boot.

Returns Pointer to info for partition structure, or NULL if no partition is found or flash read operation failed. Returned pointer is valid for the lifetime of the application.

const *esp_partition_t* ***esp_ota_get_next_update_partition**(const *esp_partition_t* *start_from)

Return the next OTA app partition which should be written with a new firmware.

Call this function to find an OTA app partition which can be passed to esp_ota_begin().

Finds next partition round-robin, starting from the current running partition.

Parameters **start_from** –If set, treat this partition info as describing the current running partition. Can be NULL, in which case esp_ota_get_running_partition() is used to find the currently running partition. The result of this function is never the same as this argument.

Returns Pointer to info for partition which should be updated next. NULL result indicates invalid OTA data partition, or that no eligible OTA app slot partition was found.

esp_err_t **esp_ota_get_partition_description**(const *esp_partition_t* *partition, *esp_app_desc_t* *app_desc)

Returns esp_app_desc structure for app partition. This structure includes app version.

Returns a description for the requested app partition.

Parameters
- **partition** –**[in]** Pointer to app partition. (only app partition)
- **app_desc** –**[out]** Structure of info about app.

Returns
- ESP_OK Successful.
- ESP_ERR_NOT_FOUND app_desc structure is not found. Magic word is incorrect.
- ESP_ERR_NOT_SUPPORTED Partition is not application.
- ESP_ERR_INVALID_ARG Arguments is NULL or if partition ' s offset exceeds partition size.
- ESP_ERR_INVALID_SIZE Read would go out of bounds of the partition.
- or one of error codes from lower-level flash driver.

uint8_t **esp_ota_get_app_partition_count**(void)

Returns number of ota partitions provided in partition table.

Returns
- Number of OTA partitions

esp_err_t **esp_ota_mark_app_valid_cancel_rollback**(void)

This function is called to indicate that the running app is working well.

Returns
- ESP_OK: if successful.

esp_err_t **esp_ota_mark_app_invalid_rollback_and_reboot**(void)

This function is called to roll back to the previously workable app with reboot.

If rollback is successful then device will reset else API will return with error code. Checks applications on a flash drive that can be booted in case of rollback. If the flash does not have at least one app (except the running app) then rollback is not possible.

Returns
- ESP_FAIL: if not successful.
- ESP_ERR_OTA_ROLLBACK_FAILED: The rollback is not possible due to flash does not have any apps.

const *esp_partition_t* ***esp_ota_get_last_invalid_partition**(void)

Returns last partition with invalid state (ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED).

Returns partition.

esp_err_t **esp_ota_get_state_partition**(const *esp_partition_t* *partition, esp_ota_img_states_t *ota_state)

Returns state for given partition.

Parameters
- **partition** –**[in]** Pointer to partition.
- **ota_state** –**[out]** state of partition (if this partition has a record in otadata).

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: partition or ota_state arguments were NULL.
- ESP_ERR_NOT_SUPPORTED: partition is not ota.
- ESP_ERR_NOT_FOUND: Partition table does not have otadata or state was not found for given partition.

esp_err_t **esp_ota_erase_last_boot_app_partition**(void)

Erase previous boot app partition and corresponding otadata select for this partition.

When current app is marked to as valid then you can erase previous app partition.

Returns

- ESP_OK: Successful, otherwise ESP_ERR.

bool **esp_ota_check_rollback_is_possible**(void)

Checks applications on the slots which can be booted in case of rollback.

These applications should be valid (marked in otadata as not UNDEFINED, INVALID or ABORTED and crc is good) and be able booted, and secure_version of app >= secure_version of efuse (if anti-rollback is enabled).

Returns

- True: Returns true if the slots have at least one app (except the running app).
- False: The rollback is not possible.

Macros

OTA_SIZE_UNKNOWN

Used for esp_ota_begin() if new image size is unknown

OTA_WITH_SEQUENTIAL_WRITES

Used for esp_ota_begin() if new image size is unknown and erase can be done in incremental manner (assuming write operation is in continuous sequence)

ESP_ERR_OTA_BASE

Base error code for ota_ops api

ESP_ERR_OTA_PARTITION_CONFLICT

Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID

Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED

Error if OTA app image is invalid

ESP_ERR_OTA_SMALL_SEC_VER

Error if the firmware has a secure version less than the running firmware.

ESP_ERR_OTA_ROLLBACK_FAILED

Error if flash does not have valid firmware in passive partition and hence rollback is not possible

ESP_ERR_OTA_ROLLBACK_INVALID_STATE

Error if current active firmware is still marked in pending validation state (ESP_OTA_IMG_PENDING_VERIFY), essentially first boot of firmware image post upgrade and hence firmware upgrade is not possible

Type Definitions

typedef uint32_t **esp_ota_handle_t**

Opaque handle for an application OTA update.

esp_ota_begin() returns a handle which is then used for subsequent calls to esp_ota_write() and esp_ota_end().

Debugging OTA Failure

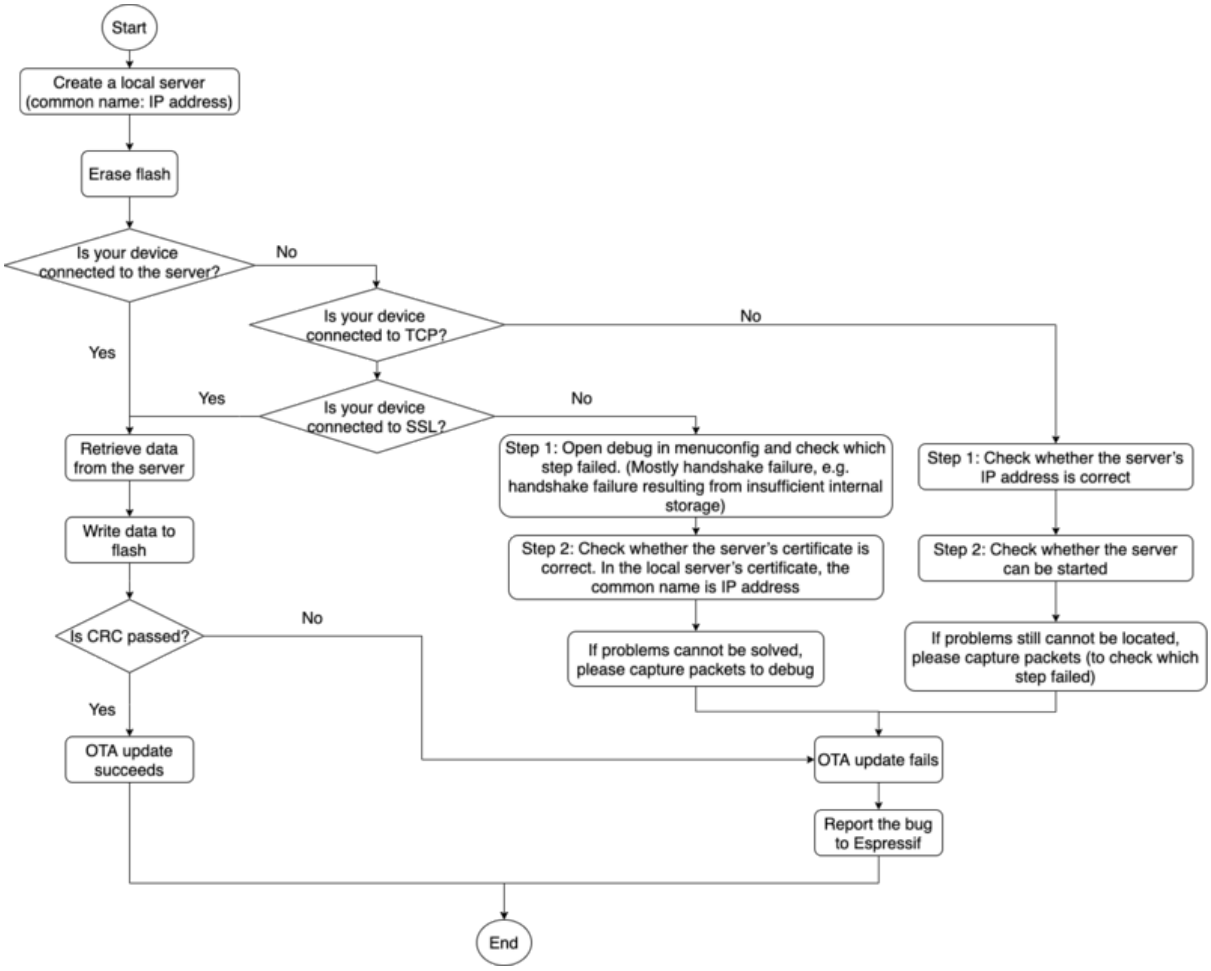


Fig. 40: How to Debug When OTA Fails (click to enlarge)

2.10.22 Performance Monitor

The Performance Monitor component provides APIs to use ESP32 internal performance counters to profile functions and applications.

Application Example

An example which combines performance monitor is provided in `examples/system/perfmon` directory. This example initializes the performance monitor structure and execute them with printing the statistics.

High level API Reference

Header Files

- perfmon/include/perfmon.h

API Reference

Header File

- components/perfmon/include/xtensa_perfmon_access.h

Functions

esp_err_t **xtensa_perfmon_init**(int id, uint16_t select, uint16_t mask, int kernelcnt, int tracelevel)

Init Performance Monitoor.

Initialize performance monitor register with define values

Parameters
- **id** –**[in]** performance counter number
- **select** –**[in]** select value from PMCTRLx register
- **mask** –**[in]** mask value from PMCTRLx register
- **kernelcnt** –**[in]** kernelcnt value from PMCTRLx register
- **tracelevel** –**[in]** tracelevel value from PMCTRLx register

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if one of the arguments is not correct

esp_err_t **xtensa_perfmon_reset**(int id)

Reset PM counter.

Reset PM counter. Writes 0 to the PMx register.

Parameters **id** –**[in]** performance counter number

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if id out of range

void **xtensa_perfmon_start**(void)

Start PM counters.

Start all PM counters synchronously. Write 1 to the PGM register

void **xtensa_perfmon_stop**(void)

Stop PM counters.

Stop all PM counters synchronously. Write 0 to the PGM register

uint32_t **xtensa_perfmon_value**(int id)

Read PM counter.

Read value of defined PM counter.

Parameters **id** –**[in]** performance counter number

Returns
- Performance counter value

esp_err_t **xtensa_perfmon_overflow**(int id)

Read PM overflow state.

Read overflow value of defined PM counter.

Parameters **id** –**[in]** performance counter number

Returns
- ESP_OK if there is no overflow (overflow = 0)
- ESP_FAIL if overflow occure (overflow = 1)

void **xtensa_perfmon_dump**(void)

Dump PM values.

Dump all PM register to the console.

Header File

- components/perfmon/include/xtensa_perfmon_apis.h

Functions

esp_err_t **xtensa_perfmon_exec**(const *xtensa_perfmon_config_t* *config)

Execute PM.

Execute performance counter for dedicated function with defined parameters

Parameters **config** –**[in]** pointer to the configuration structure

Returns

- ESP_OK if no errors
- ESP_ERR_INVALID_ARG if one of the required parameters not defined
- ESP_FAIL - counter overflow

void **xtensa_perfmon_view_cb**(void *params, uint32_t select, uint32_t mask, uint32_t value)

Dump PM results.

Callback to dump perfmon result to a FILE* stream specified in perfmon_config_t::callback_params. If callback_params is set to NULL, will print to stdout

Parameters

- **params** –**[in]** used parameters passed from configuration (callback_params). This parameter expected as FILE* hanle, where data will be stored. If this parameter NULL, then data will be stored to the stdout.
- **select** –**[in]** select value for current counter
- **mask** –**[in]** mask value for current counter
- **value** –**[in]** counter value for current counter

Structures

struct **xtensa_perfmon_config**

Performance monitor configuration structure.

Structure to configure performance counter to measure dedicated function

Public Members

int **repeat_count**

how much times function will be called before the calback will be repeated

float **max_deviation**

Difference between min and max counter number 0..1, 0 - no difference, 1 - not used

void ***call_params**

This pointer will be passed to the call_function as a parameter

void (***call_function**)(void *params)

pointer to the function that have to be called

void (***callback**)(void *params, uint32_t select, uint32_t mask, uint32_t value)

pointer to the function that will be called with result parameters

void ***callback_params**

parameter that will be passed to the callback

int **tracelevel**

trace level for all counters. In case of negative value, the filter will be ignored. If it's >=0, then the perfmon will count only when interrupt level > tracelevel. It's useful to monitor interrupts.

uint32_t **counters_size**

amount of counter in the list

const uint32_t ***select_mask**

list of the select/mask parameters

Type Definitions

typedef struct *xtensa_perfmon_config* **xtensa_perfmon_config_t**

Performance monitor configuration structure.

Structure to configure performance counter to measure dedicated function

2.10.23 Power Management

Overview

Power management algorithm included in ESP-IDF can adjust the advanced peripheral bus (APB) frequency, CPU frequency, and put the chip into light sleep mode to run an application at smallest possible power consumption, given the requirements of application components.

Application components can express their requirements by creating and acquiring power management locks.

For example:

- Driver for a peripheral clocked from APB can request the APB frequency to be set to 80 MHz while the peripheral is used.
- RTOS can request the CPU to run at the highest configured frequency while there are tasks ready to run.
- A peripheral driver may need interrupts to be enabled, which means it will have to request disabling light sleep.

Since requesting higher APB or CPU frequencies or disabling light sleep causes higher current consumption, please keep the usage of power management locks by components to a minimum.

Configuration

Power management can be enabled at compile time, using the option *CONFIG_PM_ENABLE*.

Enabling power management features comes at the cost of increased interrupt latency. Extra latency depends on a number of factors, such as the CPU frequency, single/dual core mode, whether or not frequency switch needs to be done. Minimum extra latency is 0.2 us (when the CPU frequency is 240 MHz and frequency scaling is not enabled). Maximum extra latency is 40 us (when frequency scaling is enabled, and a switch from 40 MHz to 80 MHz is performed on interrupt entry).

Dynamic frequency scaling (DFS) and automatic light sleep can be enabled in an application by calling the function `esp_pm_configure()`. Its argument is a structure defining the frequency scaling settings, `esp_pm_config_esp32_t`. In this structure, three fields need to be initialized:

- `max_freq_mhz`: Maximum CPU frequency in MHz, i.e., the frequency used when the `ESP_PM_CPU_FREQ_MAX` lock is acquired. This field will usually be set to the default CPU frequency.
- `min_freq_mhz`: Minimum CPU frequency in MHz, i.e., the frequency used when only the `ESP_PM_APB_FREQ_MAX` lock is acquired. This field can be set to the XTAL frequency value, or the XTAL frequency divided by an integer. Note that 10 MHz is the lowest frequency at which the default REF_TICK clock of 1 MHz can be generated.

- `light_sleep_enable`: Whether the system should automatically enter light sleep when no locks are acquired (`true`/`false`).
  Alternatively, if you enable the option *CONFIG_PM_DFS_INIT_AUTO* in menuconfig, the maximum CPU frequency will be determined by the *CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ* setting, and the minimum CPU frequency will be locked to the XTAL frequency.

Note: Automatic light sleep is based on FreeRTOS Tickless Idle functionality. If automatic light sleep is requested while the option *CONFIG_FREERTOS_USE_TICKLESS_IDLE* is not enabled in menuconfig, *esp_pm_configure()* will return the error *ESP_ERR_NOT_SUPPORTED*.

Note: In light sleep, peripherals are clock gated, and interrupts (from GPIOs and internal peripherals) will not be generated. A wakeup source described in the *Sleep Modes* documentation can be used to trigger wakeup from the light sleep state.

For example, the EXT0 and EXT1 wakeup sources can be used to wake up the chip via a GPIO.

Power Management Locks

Applications have the ability to acquire/release locks in order to control the power management algorithm. When an application acquires a lock, the power management algorithm operation is restricted in a way described below. When the lock is released, such restrictions are removed.

Power management locks have acquire/release counters. If the lock has been acquired a number of times, it needs to be released the same number of times to remove associated restrictions.

ESP32 supports three types of locks described in the table below.

Lock	Description
`ESP_PM_CPU_FREQ_MAX`	Requests CPU frequency to be at the maximum value set with *esp_pm_configure()*. For ESP32, this value can be set to 80 MHz, 160 MHz, or 240 MHz.
`ESP_PM_APB_FREQ_MAX`	Requests the APB frequency to be at the maximum supported value. For ESP32, this is 80 MHz.
`ESP_PM_NO_LIGHT_SLEEP`	Disables automatic switching to light sleep.

ESP32 Power Management Algorithm

The table below shows how CPU and APB frequencies will be switched if dynamic frequency scaling is enabled. You can specify the maximum CPU frequency with either *esp_pm_configure()* or *CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ*.

Max CPU Frequency Set	Lock Acquisition	CPU and APB Frequencies
240	Any of `ESP_PM_CPU_FREQ_MAX` or `ESP_PM_APB_FREQ_MAX` acquired	CPU: 240 MHz APB: 80 MHz
	None	Min values for both frequencies set with *esp_pm_configure()*
160	`ESP_PM_CPU_FREQ_MAX` acquired	CPU: 160 MHz APB: 80 MHz
	`ESP_PM_APB_FREQ_MAX` acquired, `ESP_PM_CPU_FREQ_MAX` not acquired	CPU: 80 MHz APB: 80 MHz
	None	Min values for both frequencies set with *esp_pm_configure()*
80	Any of `ESP_PM_CPU_FREQ_MAX` or `ESP_PM_APB_FREQ_MAX` acquired	CPU: 80 MHz APB: 80 MHz
	None	Min values for both frequencies set with *esp_pm_configure()*

If none of the locks are acquired, and light sleep is enabled in a call to *esp_pm_configure()*, the system will go into light sleep mode. The duration of light sleep will be determined by:

- FreeRTOS tasks blocked with finite timeouts
- Timers registered with *High resolution timer* APIs

Light sleep duration will be chosen to wake up the chip before the nearest event (task being unblocked, or timer elapses).

To skip unnecessary wake-up, you can consider initializing an esp_timer with the *skip_unhandled_events* option as true. Timers with this flag will not wake up the system and it helps to reduce consumption.

Dynamic Frequency Scaling and Peripheral Drivers

When DFS is enabled, the APB frequency can be changed multiple times within a single RTOS tick. The APB frequency change does not affect the operation of some peripherals, while other peripherals may have issues. For example, Timer Group peripheral timers will keep counting, however, the speed at which they count will change proportionally to the APB frequency.

Peripheral clock sources such as `REF_TICK`, `XTAL`, `RC_FAST` (i.e. `RTC_8M`), their frequencies will not be influenced by APB frequency. And therefore, to ensure the peripheral behaves consistently during DFS, it is recommanded to select one of these clocks as the peripheral clock source. For more specific guidelines, please refer to the "Power Management" section of each peripheral's "API Reference > Peripherals API" page.

Currently, the following peripheral drivers are aware of DFS and will use the `ESP_PM_APB_FREQ_MAX` lock for the duration of the transaction:

- SPI master

- I2C
- I2S (If the APLL clock is used, then it will use the `ESP_PM_NO_LIGHT_SLEEP` lock)
- SDMMC

The following drivers will hold the `ESP_PM_APB_FREQ_MAX` lock while the driver is enabled:

- **SPI slave**: between calls to *`spi_slave_initialize()`* and *`spi_slave_free()`*.
- **GPTimer**: between calls to *`gptimer_enable()`* and *`gptimer_disable()`*.
- **Ethernet**: between calls to *`esp_eth_driver_install()`* and *`esp_eth_driver_uninstall()`*.
- **WiFi**: between calls to *`esp_wifi_start()`* and *`esp_wifi_stop()`*. If modem sleep is enabled, the lock will be released for the periods of time when radio is disabled.
- **TWAI**: between calls to *`twai_driver_install()`* and *`twai_driver_uninstall()`*.
- **Bluetooth**: between calls to *`esp_bt_controller_enable()`* and *`esp_bt_controller_disable()`*. If Bluetooth modem sleep is enabled, the `ESP_PM_APB_FREQ_MAX` lock will be released for the periods of time when radio is disabled. However the `ESP_PM_NO_LIGHT_SLEEP` lock will still be held, unless *CONFIG_BTDM_CTRL_LOW_POWER_CLOCK* option is set to "External 32kHz crystal".

The following peripheral drivers are not aware of DFS yet. Applications need to acquire/release locks themselves, when necessary:

- PCNT
- Sigma-delta
- The legacy timer group driver
- MCPWM

API Reference

Header File

- components/esp_pm/include/esp_pm.h

Functions

esp_err_t **esp_pm_configure**(const void *config)

Set implementation-specific power management configuration.

Parameters **config** – pointer to implementation-specific configuration structure (e.g. esp_pm_config_esp32)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the configuration values are not correct
- ESP_ERR_NOT_SUPPORTED if certain combination of values is not supported, or if CONFIG_PM_ENABLE is not enabled in sdkconfig

esp_err_t **esp_pm_get_configuration**(void *config)

Get implementation-specific power management configuration.

Parameters **config** – pointer to implementation-specific configuration structure (e.g. esp_pm_config_esp32)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the pointer is null

esp_err_t **esp_pm_lock_create**(*esp_pm_lock_type_t* lock_type, int arg, const char *name, *esp_pm_lock_handle_t* *out_handle)

Initialize a lock handle for certain power management parameter.

When lock is created, initially it is not taken. Call esp_pm_lock_acquire to take the lock.

This function must not be called from an ISR.

Parameters

- **lock_type** –Power management constraint which the lock should control
- **arg** –argument, value depends on lock_type, see esp_pm_lock_type_t
- **name** –arbitrary string identifying the lock (e.g. "wifi" or "spi"). Used by the esp_pm_dump_locks function to list existing locks. May be set to NULL. If not set to NULL, must point to a string which is valid for the lifetime of the lock.
- **out_handle** –**[out]** handle returned from this function. Use this handle when calling esp_pm_lock_delete, esp_pm_lock_acquire, esp_pm_lock_release. Must not be NULL.

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if the lock structure can not be allocated
- ESP_ERR_INVALID_ARG if out_handle is NULL or type argument is not valid
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

esp_err_t **esp_pm_lock_acquire**(*esp_pm_lock_handle_t* handle)

Take a power management lock.

Once the lock is taken, power management algorithm will not switch to the mode specified in a call to esp_pm_lock_create, or any of the lower power modes (higher numeric values of 'mode').

The lock is recursive, in the sense that if esp_pm_lock_acquire is called a number of times, esp_pm_lock_release has to be called the same number of times in order to release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other esp_pm_lock_* functions for the same handle.

Parameters **handle** –handle obtained from esp_pm_lock_create function

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

esp_err_t **esp_pm_lock_release**(*esp_pm_lock_handle_t* handle)

Release the lock taken using esp_pm_lock_acquire.

Call to this functions removes power management restrictions placed when taking the lock.

Locks are recursive, so if esp_pm_lock_acquire is called a number of times, esp_pm_lock_release has to be called the same number of times in order to actually release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other esp_pm_lock_* functions for the same handle.

Parameters **handle** –handle obtained from esp_pm_lock_create function

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if lock is not acquired
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

esp_err_t **esp_pm_lock_delete**(*esp_pm_lock_handle_t* handle)

Delete a lock created using esp_pm_lock.

The lock must be released before calling this function.

This function must not be called from an ISR.

Parameters **handle** –handle obtained from esp_pm_lock_create function

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle argument is NULL
- ESP_ERR_INVALID_STATE if the lock is still acquired

- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

esp_err_t **esp_pm_dump_locks**(FILE *stream)

Dump the list of all locks to stderr

This function dumps debugging information about locks created using esp_pm_lock_create to an output stream.

This function must not be called from an ISR. If esp_pm_lock_acquire/release are called while this function is running, inconsistent results may be reported.

Parameters **stream** – stream to print information to; use stdout or stderr to print to the console; use fmemopen/open_memstream to print to a string buffer.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

Type Definitions

typedef struct esp_pm_lock ***esp_pm_lock_handle_t**

Opaque handle to the power management lock.

Enumerations

enum **esp_pm_lock_type_t**

Power management constraints.

Values:

enumerator **ESP_PM_CPU_FREQ_MAX**

Require CPU frequency to be at the maximum value set via esp_pm_configure. Argument is unused and should be set to 0.

enumerator **ESP_PM_APB_FREQ_MAX**

Require APB frequency to be at the maximum value supported by the chip. Argument is unused and should be set to 0.

enumerator **ESP_PM_NO_LIGHT_SLEEP**

Prevent the system from going into light sleep. Argument is unused and should be set to 0.

Header File

- components/esp_pm/include/esp32/pm.h

Structures

struct **esp_pm_config_esp32_t**

Power management config for ESP32.

Pass a pointer to this structure as an argument to esp_pm_configure function.

Public Members

int **max_freq_mhz**

Maximum CPU frequency, in MHz

int **min_freq_mhz**

Minimum CPU frequency to use when no locks are taken, in MHz

bool **light_sleep_enable**

Enter light sleep when no locks are taken

2.10.24 POSIX Threads Support

Overview

ESP-IDF is based on FreeRTOS but offers a range of POSIX-compatible APIs that allow easy porting of third party code. This includes support for common parts of the POSIX Threads "pthreads" API.

POSIX Threads are implemented in ESP-IDF as wrappers around equivalent FreeRTOS features. The runtime memory or performance overhead of using the pthreads API is quite low, but not every feature available in either pthreads or FreeRTOS is available via the ESP-IDF pthreads support.

Pthreads can be used in ESP-IDF by including standard `pthread.h` header, which is included in the toolchain libc. An additional ESP-IDF specific header, `esp_pthread.h`, provides additional non-POSIX APIs for using some ESP-IDF features with pthreads.

C++ Standard Library implementations for `std::thread`, `std::mutex`, `std::condition_variable`, etc. are implemented using pthreads (via GCC libstdc++). Therefore, restrictions mentioned here also apply to the equivalent C++ standard library functionality.

RTOS Integration

Unlike many operating systems using POSIX Threads, ESP-IDF is a real-time operating system with a real-time scheduler. This means that a thread will only stop running if a higher priority task is ready to run, the thread blocks on an OS synchronization structure like a mutex, or the thread calls any of the functions `sleep`, *`vTaskDelay()`*, or `usleep`.

Note: If calling a standard libc or C++ sleep function, such as `usleep` defined in `unistd.h`, then the task will only block and yield the CPU if the sleep time is longer than *one FreeRTOS tick period*. If the time is shorter, the thread will busy-wait instead of yielding to another RTOS task.

By default, all POSIX Threads have the same RTOS priority, but it is possible to change this by calling a *custom API*.

Standard features

The following standard APIs are implemented in ESP-IDF.

Refer to standard POSIX Threads documentation, or pthread.h, for details about the standard arguments and behaviour of each function. Differences or limitations compared to the standard APIs are noted below.

Thread APIs

- `pthread_create()` - The `attr` argument is supported for setting stack size and detach state only. Other attribute fields are ignored. - Unlike FreeRTOS task functions, the `start_routine` function is allowed to return. A "detached" type thread is automatically deleted if the function returns. The default "joinable" type thread will be suspended until pthread_join() is called on it.
- `pthread_join()`
- `pthread_detach()`
- `pthread_exit()`
- `sched_yield()`

- `pthread_self()` - An assert will fail if this function is called from a FreeRTOS task which is not a pthread.
- `pthread_equal()`

Thread Attributes

- `pthread_attr_init()`
- `pthread_attr_destroy()` - This function doesn't need to free any resources and instead resets the `attr` structure to defaults (implementation is same as `pthread_attr_init()`).
- `pthread_attr_getstacksize()` / `pthread_attr_setstacksize()`
- `pthread_attr_getdetachstate()` / `pthread_attr_setdetachstate()`

Once

- `pthread_once()`

Static initializer constant `PTHREAD_ONCE_INIT` is supported.

Note: This function can be called from tasks created using either pthread or FreeRTOS APIs

---

Mutexes POSIX Mutexes are implemented as FreeRTOS Mutex Semaphores (normal type for "fast" or "error check" mutexes, and Recursive type for "recursive" mutexes). This means that they have the same priority inheritance behaviour as mutexes created with `xSemaphoreCreateMutex()`.

- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_timedlock()`
- `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`
- `pthread_mutexattr_init()`
- `pthread_mutexattr_destroy()`
- `pthread_mutexattr_gettype()` / `pthread_mutexattr_settype()`

Static initializer constant `PTHREAD_MUTEX_INITIALIZER` is supported, but the non-standard static initializer constants for other mutex types are not supported.

Note: These functions can be called from tasks created using either pthread or FreeRTOS APIs

---

Condition Variables

- `pthread_cond_init()` - The `attr` argument is not implemented and is ignored.
- `pthread_cond_destroy()`
- `pthread_cond_signal()`
- `pthread_cond_broadcast()`
- `pthread_cond_wait()`
- `pthread_cond_timedwait()`

Static initializer constant `PTHREAD_COND_INITIALIZER` is supported.

- The resolution of `pthread_cond_timedwait()` timeouts is the RTOS tick period (see *CONFIG_FREERTOS_HZ*). Timeouts may be delayed up to one tick period after the requested timeout.

Note: These functions can be called from tasks created using either pthread or FreeRTOS APIs

---

Read/Write Locks

- `pthread_rwlock_init()` - The `attr` argument is not implemented and is ignored.
- `pthread_rwlock_destroy()`
- `pthread_rwlock_rdlock()`
- `pthread_rwlock_wrlock()`
- `pthread_rwlock_unlock()`

Static initializer constant `PTHREAD_RWLOCK_INITIALIZER` is supported.

Note: These functions can be called from tasks created using either pthread or FreeRTOS APIs

---

Thread-Specific Data

- `pthread_key_create()` - The `destr_function` argument is supported and will be called if a thread function exits normally, calls `pthread_exit()`, or if the underlying task is deleted directly using the FreeRTOS function *`vTaskDelete()`*.
- `pthread_key_delete()`
- `pthread_setspecific()` / `pthread_getspecific()`

Note: These functions can be called from tasks created using either pthread or FreeRTOS APIs

---

Note: There are other options for thread local storage in ESP-IDF, including options with higher performance. See *Thread Local Storage*.

Not Implemented

The `pthread.h` header is a standard header and includes additional APIs and features which are not implemented in ESP-IDF. These include:

- `pthread_cancel()` returns `ENOSYS` if called.
- `pthread_condattr_init()` returns `ENOSYS` if called.

Other POSIX Threads functions (not listed here) are not implemented and will produce either a compiler or a linker error if referenced from an ESP-IDF application. If you identify a useful API that you would like to see implemented in ESP-IDF, please open a *feature request on GitHub <https://github.com/espressif/esp-idf/issues>* with the details.

ESP-IDF Extensions

The API *`esp_pthread_set_cfg()`* defined in the `esp_pthreads.h` header offers custom extensions to control how subsequent calls to `pthread_create()` will behave. Currently, the following configuration can be set:

- Default stack size of new threads, if not specified when calling `pthread_create()` (overrides *CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT*).
- RTOS priority of new threads (overrides *CONFIG_PTHREAD_TASK_PRIO_DEFAULT*).
- Core affinity / core pinning of new threads (overrides *CONFIG_PTHREAD_TASK_CORE_DEFAULT*).
- FreeRTOS task name for new threads (overrides *CONFIG_PTHREAD_TASK_NAME_DEFAULT*)

This configuration is scoped to the calling thread (or FreeRTOS task), meaning that *`esp_pthread_set_cfg()`* can be called independently in different threads or tasks. If the `inherit_cfg` flag is set in the current configuration then any new thread created will inherit the creator's configuration (if that thread calls `pthread_create()` recursively), otherwise the new thread will have the default configuration.

Examples

- system/pthread demonstrates using the pthreads API to create threads
- cxx/pthread demonstrates using C++ Standard Library functions with threads

API Reference

Header File

- components/pthread/include/esp_pthread.h

Functions

esp_pthread_cfg_t **esp_pthread_get_default_config**(void)

Creates a default pthread configuration based on the values set via menuconfig.

Returns A default configuration structure.

esp_err_t **esp_pthread_set_cfg**(const *esp_pthread_cfg_t* *cfg)

Configure parameters for creating pthread.

This API allows you to configure how the subsequent pthread_create() call will behave. This call can be used to setup configuration parameters like stack size, priority, configuration inheritance etc.

If the 'inherit' flag in the configuration structure is enabled, then the same configuration is also inherited in the thread subtree.

Note: Passing non-NULL attributes to pthread_create() will override the stack_size parameter set using this API

Parameters **cfg** –The pthread config parameters

Returns

- ESP_OK if configuration was successfully set
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_ARG if stack_size is less than PTHREAD_STACK_MIN

esp_err_t **esp_pthread_get_cfg**(*esp_pthread_cfg_t* *p)

Get current pthread creation configuration.

This will retrieve the current configuration that will be used for creating threads.

Parameters **p** –Pointer to the pthread config structure that will be updated with the currently configured parameters

Returns

- ESP_OK if the configuration was available
- ESP_ERR_NOT_FOUND if a configuration wasn't previously set

esp_err_t **esp_pthread_init**(void)

Initialize pthread library.

Structures

struct **esp_pthread_cfg_t**

pthread configuration structure that influences pthread creation

Public Members

size_t **stack_size**

The stack size of the pthread.

size_t **prio**

The thread' s priority.

bool **inherit_cfg**

Inherit this configuration further.

const char ***thread_name**

The thread name.

int **pin_to_core**

The core id to pin the thread to. Has the same value range as xCoreId argument of xTaskCreatePinnedToCore.

Macros

PTHREAD_STACK_MIN

2.10.25 Random Number Generation

ESP32 contains a hardware random number generator, values from it can be obtained using the APIs `esp_random()` and `esp_fill_random()`.

The hardware RNG produces true random numbers under any of the following conditions:

- RF subsystem is enabled (i.e. Wi-Fi or Bluetooth are enabled).
- An internal entropy source has been enabled by calling `bootloader_random_enable()` and not yet disabled by calling `bootloader_random_disable()`.
- While the ESP-IDF *Second stage bootloader* is running. This is because the default ESP-IDF bootloader implementation calls `bootloader_random_enable()` when the bootloader starts, and `bootloader_random_disable()` before executing the app.

When any of these conditions are true, samples of physical noise are continuously mixed into the internal hardware RNG state to provide entropy. Consult the *ESP32 Technical Reference Manual* > *Random Number Generator (RNG)* [PDF] chapter for more details.

If none of the above conditions are true, the output of the RNG should be considered pseudo-random only.

Startup

During startup, ESP-IDF bootloader temporarily enables a non-RF entropy source (internal reference voltage noise) that provides entropy for any first boot key generation. However, after the app starts executing then normally only pseudo-random numbers are available until Wi-Fi or Bluetooth are initialized.

To re-enable the entropy source temporarily during app startup, or for an application that does not use Wi-Fi or Bluetooth, call the function `bootloader_random_enable()` to re-enable the internal entropy source. The function `bootloader_random_disable()` must be called to disable the entropy source again before using ADC, I2S, Wi-Fi or Bluetooth.

Note: The entropy source enabled during the boot process by the ESP-IDF Second Stage Bootloader will seed the internal RNG state with some entropy. However, the internal hardware RNG state is not large enough to provide a

continuous stream of true random numbers. This is why a continuous entropy source must be enabled whenever true random numbers are required.

Note: If an application requires a source of true random numbers but it is not possible to permanently enable a hardware entropy source, consider using a strong software DRBG implementation such as the mbedTLS CTR-DRBG or HMAC-DRBG, with an initial seed of entropy from hardware RNG true random numbers.

API Reference

Header File

- components/esp_hw_support/include/esp_random.h

Functions

uint32_t **esp_random**(void)

Get one random 32-bit word from hardware RNG.

If Wi-Fi or Bluetooth are enabled, this function returns true random numbers. In other situations, if true random numbers are required then consult the ESP-IDF Programming Guide "Random Number Generation" section for necessary prerequisites.

This function automatically busy-waits to ensure enough external entropy has been introduced into the hardware RNG state, before returning a new random number. This delay is very short (always less than 100 CPU cycles).

Returns Random value between 0 and UINT32_MAX

void **esp_fill_random**(void *buf, size_t len)

Fill a buffer with random bytes from hardware RNG.

Note: This function is implemented via calls to esp_random(), so the same constraints apply.

Parameters

- **buf** –Pointer to buffer to fill with random numbers.
- **len** –Length of buffer in bytes

Header File

- components/bootloader_support/include/bootloader_random.h

Functions

void **bootloader_random_enable**(void)

Enable an entropy source for RNG if RF subsystem is disabled.

The exact internal entropy source mechanism depends on the chip in use but all SoCs use the SAR ADC to continuously mix random bits (an internal noise reading) into the HWRNG. Consult the SoC Technical Reference Manual for more information.

Can also be called from app code, if true random numbers are required without initialized RF subsystem. This might be the case in early startup code of the application when the RF subsystem has not started yet or if the RF subsystem should not be enabled for power saving.

Consult ESP-IDF Programming Guide "Random Number Generation" section for details.

> **Warning:** This function is not safe to use if any other subsystem is accessing the RF subsystem or the ADC at the same time!

void **bootloader_random_disable**(void)

Disable entropy source for RNG.

Disables internal entropy source. Must be called after bootloader_random_enable() and before RF subsystem features, ADC, or I2S (ESP32 only) are initialized.

Consult the ESP-IDF Programming Guide "Random Number Generation" section for details.

void **bootloader_fill_random**(void *buffer, size_t length)

Fill buffer with 'length' random bytes.

Note: If this function is being called from app code only, and never from the bootloader, then it's better to call esp_fill_random().

Parameters

- **buffer** –Pointer to buffer
- **length** –This many bytes of random data will be copied to buffer

getrandom

A compatible version of the Linux `getrandom()` function is also provided for ease of porting:

```
#include <sys/random.h>

ssize_t getrandom(void *buf, size_t buflen, unsigned int flags);
```

This function is implemented by calling *esp_fill_random()* internally.

The `flags` argument is ignored, this function is always non-blocking but the strength of any random numbers is dependent on the same conditions described above.

Return value is -1 (with `errno` set to `EFAULT`) if the `buf` argument is NULL, and equal to `buflen` otherwise.

2.10.26 Sleep Modes

Overview

ESP32 contains the following power saving modes: Light-sleep, and Deep-sleep.

In Light-sleep mode, the digital peripherals, most of the RAM, and CPUs are clock-gated and their supply voltage is reduced. Upon exit from Light-sleep, the digital peripherals, RAM, and CPUs resume operation and their internal states are preserved.

In Deep-sleep mode, the CPUs, most of the RAM, and all digital peripherals that are clocked from APB_CLK are powered off. The only parts of the chip that remain powered on are:

- RTC controller
- ULP coprocessor
- RTC fast memory
- RTC slow memory

There are several wakeup sources in Deep-sleep and Light-sleep modes. These sources can also be combined so that the chip will wake up when any of the sources are triggered. Wakeup sources can be enabled using `esp_sleep_enable_X_wakeup` APIs and can be disabled using `esp_sleep_disable_wakeup_source()` API. Next section describes these APIs in detail. Wakeup sources can be configured at any moment before entering Light-sleep or Deep-sleep mode.

Additionally, the application can force specific powerdown modes for RTC peripherals and RTC memories using `esp_sleep_pd_config()` API.

Once wakeup sources are configured, the application can enter sleep mode using `esp_light_sleep_start()` or `esp_deep_sleep_start()` APIs. At this point, the hardware will be configured according to the requested wakeup sources, and the RTC controller will either power down or power off the CPUs and digital peripherals.

Wi-Fi/Bluetooth and Sleep Modes

In Deep-sleep and Light-sleep modes, the wireless peripherals are powered down. Before entering Light-sleep modes, the application must disable Wi-Fi and Bluetooth using the appropriate calls (i.e., `esp_bluedroid_disable()`, `esp_bt_controller_disable()`, `esp_wifi_stop()`). In both Deep-sleep and Light-sleep modes, Wi-Fi and Bluetooth connection cannot be maintained.

If Wi-Fi/Bluetooth connections need to be maintained, enable Wi-Fi/Bluetooth Modem-sleep mode and automatic Light-sleep feature (see *Power Management APIs*). This will allow the system to wake up from sleep automatically when required by the Wi-Fi/Bluetooth driver, thereby maintaining the connection.

Wakeup Sources

Timer The RTC controller has a built-in timer which can be used to wake up the chip after a predefined amount of time. Time is specified at microsecond precision, but the actual resolution depends on the clock source selected for RTC SLOW_CLK.

For details on RTC clock options, see *ESP32 Technical Reference Manual* > *ULP Coprocessor* [PDF].

RTC peripherals or RTC memories don't need to be powered on during sleep in this wakeup mode.

`esp_sleep_enable_timer_wakeup()` function can be used to enable sleep wakeup using a timer.

Touchpad The RTC IO module contains the logic to trigger wakeup when a touch sensor interrupt occurs. To wakeup from a touch sensor interrupt, users need to configure the touch pad interrupt before the chip enters Deep-sleep or Light-sleep modes.

Revisions 0 and 1 of ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e., ESP_PD_DOMAIN_RTC_PERIPH should be set to ESP_PD_OPTION_AUTO).

`esp_sleep_enable_touchpad_wakeup()` function can be used to enable this wakeup source.

External Wakeup (ext0) The RTC IO module contains the logic to trigger wakeup when one of RTC GPIOs is set to a predefined logic level. RTC IO is part of the RTC peripherals power domain, so RTC peripherals will be kept powered on during Deep-sleep if this wakeup source is requested.

The RTC IO module is enabled in this mode, so internal pullup or pulldown resistors can also be used. They need to be configured by the application using `rtc_gpio_pullup_en()` and `rtc_gpio_pulldown_en()` functions before calling `esp_deep_sleep_start()`.

In revisions 0 and 1 of ESP32, this wakeup source is incompatible with ULP and touch wakeup sources.

`esp_sleep_enable_ext0_wakeup()` function can be used to enable this wakeup source.

Warning: After waking up from sleep, the IO pad used for wakeup will be configured as RTC IO. Therefore, before using this pad as digital GPIO, users need to reconfigure it using `rtc_gpio_deinit()` function.

External Wakeup (ext1) The RTC controller contains the logic to trigger wakeup using multiple RTC GPIOs. One of the following two logic functions can be used to trigger wakeup:

- wake up if any of the selected pins is high (`ESP_EXT1_WAKEUP_ANY_HIGH`)
- wake up if all the selected pins are low (`ESP_EXT1_WAKEUP_ALL_LOW`)

This wakeup source is implemented by the RTC controller. As such, RTC peripherals and RTC memories can be powered down in this mode. However, if RTC peripherals are powered down, internal pullup and pulldown resistors will be disabled. To use internal pullup or pulldown resistors, request the RTC peripherals power domain to be kept on during sleep, and configure pullup/pulldown resistors using `rtc_gpio_` functions before entering sleep:

```
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_ON);
rtc_gpio_pullup_dis(gpio_num);
rtc_gpio_pulldown_en(gpio_num);
```

Warning:

- To use the EXT1 wakeup, the IO pad(s) are configured as RTC IO. Thus if these pads want to be used as digital GPIOs after waking up from sleep, *`rtc_gpio_deinit()`* needs to be called first.
- If the RTC peripherals are configured to be powered down (which is by default), the wakeup IOs will be set to holding state before entering sleep. Therefore, after waking up from Light-sleep, please call *rtc_gpio_hold_dis* to disable the hold function to perform any pin re-configuration. For Deep-sleep wakeup, this is already being handled at the application startup stage.

`esp_sleep_enable_ext1_wakeup()` function can be used to enable this wakeup source.

ULP Coprocessor Wakeup ULP coprocessor can run while the chip is in sleep mode, and may be used to poll sensors, monitor ADC or touch sensor values, and wake up the chip when a specific event is detected. ULP coprocessor is part of the RTC peripherals power domain, and it runs the program stored in RTC slow memory. RTC slow memory will be powered on during sleep if this wakeup mode is requested. RTC peripherals will be automatically powered on before ULP coprocessor starts running the program; once the program stops running, RTC peripherals are automatically powered down again.

Revisions 0 and 1 of ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e., ESP_PD_DOMAIN_RTC_PERIPH should be set to ESP_PD_OPTION_AUTO).

`esp_sleep_enable_ulp_wakeup()` function can be used to enable this wakeup source.

GPIO Wakeup (Light-sleep Only) In addition to EXT0 and EXT1 wakeup sources described above, one more method of wakeup from external inputs is available in Light-sleep mode. With this wakeup source, each pin can be individually configured to trigger wakeup on high or low level using *`gpio_wakeup_enable()`* function. Unlike EXT0 and EXT1 wakeup sources, which can only be used with RTC IOs, this wakeup source can be used with any IO (RTC or digital).

`esp_sleep_enable_gpio_wakeup()` function can be used to enable this wakeup source.

Warning: Before entering Light-sleep mode, check if any GPIO pin to be driven is part of the VDD_SDIO power domain. If so, this power domain must be configured to remain ON during sleep.

For example, on ESP32-WROOM-32 board, GPIO16 and GPIO17 are linked to VDD_SDIO power domain. If they are configured to remain high during Light-sleep, the power domain should be configured to remain powered ON. This can be done with *`esp_sleep_pd_config()`*:

```
esp_sleep_pd_config(ESP_PD_DOMAIN_VDDSDIO, ESP_PD_OPTION_ON);
```

UART Wakeup (Light-sleep Only) When ESP32 receives UART input from external devices, it is often necessary to wake up the chip when input data is available. The UART peripheral contains a feature which allows waking up

the chip from Light-sleep when a certain number of positive edges on RX pin are seen. This number of positive edges can be set using *uart_set_wakeup_threshold()* function. Note that the character which triggers wakeup (and any characters before it) will not be received by the UART after wakeup. This means that the external device typically needs to send an extra character to the ESP32 to trigger wakeup before sending the data.

esp_sleep_enable_uart_wakeup() function can be used to enable this wakeup source.

Power-down of RTC Peripherals and Memories

By default, *esp_deep_sleep_start()* and *esp_light_sleep_start()* functions will power down all RTC power domains which are not needed by the enabled wakeup sources. To override this behaviour, *esp_sleep_pd_config()* function is provided.

Note: in revision 0 of ESP32, RTC fast memory will always be kept enabled in Deep-sleep, so that the Deep-sleep stub can run after reset. This can be overridden, if the application doesn't need clean reset behaviour after Deep-sleep.

If some variables in the program are placed into RTC slow memory (for example, using `RTC_DATA_ATTR` attribute), RTC slow memory will be kept powered on by default. This can be overridden using *esp_sleep_pd_config()* function, if desired.

Power-down of Flash

By default, to avoid potential issues, *esp_light_sleep_start()* function will **not** power down flash. To be more specific, it takes time to power down the flash and during this period the system may be woken up, which then actually powers up the flash before this flash could be powered down completely. As a result, there is a chance that the flash may not work properly.

So, in theory, it's ok if you only wake up the system after the flash is completely powered down. However, in reality, the flash power-down period can be hard to predict (for example, this period can be much longer when you add filter capacitors to the flash's power supply circuit) and uncontrollable (for example, the asynchronous wake-up signals make the actual sleep time uncontrollable).

Warning: If a filter capacitor is added to your flash power supply circuit, please do everything possible to avoid powering down flash.

Therefore, it's recommended not to power down flash when using ESP-IDF. For power-sensitive applications, it's recommended to use Kconfig option *CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND* to reduce the power consumption of the flash during light sleep, instead of powering down the flash.

It is worth mentioning that PSRAM has a similar Kconfig option *CONFIG_ESP_SLEEP_PSRAM_LEAKAGE_WORKAROUND*.

However, for those who have fully understood the risk and are still willing to power down the flash to further reduce the power consumption, please check the following mechanisms:

- Setting Kconfig option *CONFIG_ESP_SLEEP_POWER_DOWN_FLASH* only powers down the flash when the RTC timer is the only wake-up source **and** the sleep time is longer than the flash power-down period.
- Calling `esp_sleep_pd_config(ESP_PD_DOMAIN_VDDSDIO, ESP_PD_OPTION_OFF)` powers down flash when the RTC timer is not enabled as a wakeup source **or** the sleep time is longer than the flash power-down period.

Note:

- ESP-IDF does not provide any mechanism that can power down the flash in all conditions when light sleep.
- *esp_deep_sleep_start()* function will force power down flash regardless of user configuration.

Entering Light-sleep

esp_light_sleep_start() function can be used to enter Light-sleep once wakeup sources are configured. It is also possible to enter Light-sleep with no wakeup sources configured. In this case, the chip will be in Light-sleep mode indefinitely until external reset is applied.

Entering Deep-sleep

esp_deep_sleep_start() function can be used to enter Deep-sleep once wakeup sources are configured. It is also possible to enter Deep-sleep with no wakeup sources configured. In this case, the chip will be in Deep-sleep mode indefinitely until external reset is applied.

Configuring IOs

Some ESP32 IOs have internal pullups or pulldowns, which are enabled by default. If an external circuit drives this pin in Deep-sleep mode, current consumption may increase due to current flowing through these pullups and pulldowns.

To isolate a pin to prevent extra current draw, call *rtc_gpio_isolate()* function.

For example, on ESP32-WROVER module, GPIO12 is pulled up externally, and it also has an internal pulldown in the ESP32 chip. This means that in Deep-sleep, some current will flow through these external and internal resistors, increasing Deep-sleep current above the minimal possible value.

Add the following code before *esp_deep_sleep_start()* to remove such extra current:

```
rtc_gpio_isolate(GPIO_NUM_12);
```

UART Output Handling

Before entering sleep mode, *esp_deep_sleep_start()* will flush the contents of UART FIFOs.

When entering Light-sleep mode using *esp_light_sleep_start()*, UART FIFOs will not be flushed. Instead, UART output will be suspended, and remaining characters in the FIFO will be sent out after wakeup from Light-sleep.

Checking Sleep Wakeup Cause

esp_sleep_get_wakeup_cause() function can be used to check which wakeup source has triggered wakeup from sleep mode.

For touchpad, it is possible to identify which touch pin has caused wakeup using *esp_sleep_get_touchpad_wakeup_status()* functions.

For ext1 wakeup sources, it is possible to identify which touch pin has caused wakeup using *esp_sleep_get_ext1_wakeup_status()* functions.

Disable Sleep Wakeup Source

Previously configured wakeup sources can be disabled later using *esp_sleep_disable_wakeup_source()* API. This function deactivates trigger for the given wakeup source. Additionally, it can disable all triggers if the argument is `ESP_SLEEP_WAKEUP_ALL`.

Application Example

- protocols/sntp: the implementation of basic functionality of Deep-sleep, where ESP module is periodically waken up to retrieve time from NTP server.
- wifi/power_save: the implementation of Wi-Fi Modem-sleep example.
- bluetooth/nimble/power_save: the implementation of Bluetooth Modem-sleep example.
- system/deep_sleep: the usage of various Deep-sleep wakeup triggers and ULP coprocessor programming.

API Reference

Header File

- components/esp_hw_support/include/esp_sleep.h

Functions

esp_err_t **esp_sleep_disable_wakeup_source**(*esp_sleep_source_t* source)

Disable wakeup source.

This function is used to deactivate wake up trigger for source defined as parameter of the function.

See docs/sleep-modes.rst for details.

Note: This function does not modify wake up configuration in RTC. It will be performed in esp_deep_sleep_start/esp_light_sleep_start function.

Parameters **source** –- number of source to disable of type esp_sleep_source_t

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if trigger was not active

esp_err_t **esp_sleep_enable_ulp_wakeup**(void)

Enable wakeup by ULP coprocessor.

Note: On ESP32, ULP wakeup source cannot be used when RTC_PERIPH power domain is forced, to be powered on (ESP_PD_OPTION_ON) or when ext0 wakeup source is used.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_RTC_EXT_CRYST_ADDIT_CURRENT) is enabled.
- ESP_ERR_INVALID_STATE if ULP co-processor is not enabled or if wakeup triggers conflict

esp_err_t **esp_sleep_enable_timer_wakeup**(uint64_t time_in_us)

Enable wakeup by timer.

Parameters **time_in_us** –time before wakeup, in microseconds

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if value is out of range (TBD)

esp_err_t **esp_sleep_enable_touchpad_wakeup**(void)

Enable wakeup by touch sensor.

Note: On ESP32, touch wakeup source can not be used when RTC_PERIPH power domain is forced to be powered on (ESP_PD_OPTION_ON) or when ext0 wakeup source is used.

Note: The FSM mode of the touch button should be configured as the timer trigger mode.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_RTC_EXT_CRYST_ADDIT_CURRENT) is enabled.
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

touch_pad_t **esp_sleep_get_touchpad_wakeup_status**(void)

Get the touch pad which caused wakeup.

If wakeup was caused by another source, this function will return TOUCH_PAD_MAX;

Returns touch pad which caused wakeup

bool **esp_sleep_is_valid_wakeup_gpio**(*gpio_num_t* gpio_num)

Returns true if a GPIO number is valid for use as wakeup source.

Note: For SoCs with RTC IO capability, this can be any valid RTC IO input pin.

Parameters **gpio_num** –Number of the GPIO to test for wakeup source capability
Returns True if this GPIO number will be accepted as a sleep wakeup source.

esp_err_t **esp_sleep_enable_ext0_wakeup**(*gpio_num_t* gpio_num, int level)

Enable wakeup using a pin.

This function uses external wakeup feature of RTC_IO peripheral. It will work only if RTC peripherals are kept on during sleep.

This feature can monitor any pin which is an RTC IO. Once the pin transitions into the state given by level argument, the chip will be woken up.

Note: This function does not modify pin configuration. The pin is configured in esp_deep_sleep_start/esp_light_sleep_start, immediately before entering sleep mode.

Note: ESP32: ext0 wakeup source can not be used together with touch or ULP wakeup sources.

Parameters

- **gpio_num** –GPIO number used as wakeup source. Only GPIOs with the RTC functionality can be used. For different SoCs, the related GPIOs are:
  - ESP32: 0, 2, 4, 12-15, 25-27, 32-39
  - ESP32-S2: 0-21
  - ESP32-S3: 0-21
- **level** –input level which will trigger wakeup (0=low, 1=high)

Returns

- ESP_OK on success

- ESP_ERR_INVALID_ARG if the selected GPIO is not an RTC GPIO, or the mode is invalid
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

esp_err_t **esp_sleep_enable_ext1_wakeup**(uint64_t mask, *esp_sleep_ext1_wakeup_mode_t* mode)

Enable wakeup using multiple pins.

This function uses external wakeup feature of RTC controller. It will work even if RTC peripherals are shut down during sleep.

This feature can monitor any number of pins which are in RTC IOs. Once any of the selected pins goes into the state given by mode argument, the chip will be woken up.

Note: This function does not modify pin configuration. The pins are configured in esp_deep_sleep_start/esp_light_sleep_start, immediately before entering sleep mode.

Note: Internal pullups and pulldowns don't work when RTC peripherals are shut down. In this case, external resistors need to be added. Alternatively, RTC peripherals (and pullups/pulldowns) may be kept enabled using esp_sleep_pd_config function.

Parameters

- **mask** –bit mask of GPIO numbers which will cause wakeup. Only GPIOs which have RTC functionality can be used in this bit map. For different SoCs, the related GPIOs are:
  - ESP32: 0, 2, 4, 12-15, 25-27, 32-39
  - ESP32-S2: 0-21
  - ESP32-S3: 0-21
  - ESP32-C6: 0-7
  - ESP32-H2: 7-14
- **mode** –select logic function used to determine wakeup condition:
  - ESP_EXT1_WAKEUP_ALL_LOW: wake up when all selected GPIOs are low
  - ESP_EXT1_WAKEUP_ANY_HIGH: wake up when any of the selected GPIOs is high

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if any of the selected GPIOs is not an RTC GPIO, or mode is invalid

esp_err_t **esp_sleep_enable_gpio_wakeup**(void)

Enable wakeup from light sleep using GPIOs.

Each GPIO supports wakeup function, which can be triggered on either low level or high level. Unlike EXT0 and EXT1 wakeup sources, this method can be used both for all IOs: RTC IOs and digital IOs. It can only be used to wakeup from light sleep though.

To enable wakeup, first call gpio_wakeup_enable, specifying gpio number and wakeup level, for each GPIO which is used for wakeup. Then call this function to enable wakeup feature.

Note: On ESP32, GPIO wakeup source can not be used together with touch or ULP wakeup sources.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

esp_err_t **esp_sleep_enable_uart_wakeup**(int uart_num)

Enable wakeup from light sleep using UART.

Use uart_set_wakeup_threshold function to configure UART wakeup threshold.

Wakeup from light sleep takes some time, so not every character sent to the UART can be received by the application.

Note: ESP32 does not support wakeup from UART2.

Parameters **uart_num** –UART port to wake up from

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if wakeup from given UART is not supported

esp_err_t **esp_sleep_enable_bt_wakeup**(void)

Enable wakeup by bluetooth.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if wakeup from bluetooth is not supported

esp_err_t **esp_sleep_disable_bt_wakeup**(void)

Disable wakeup by bluetooth.

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if wakeup from bluetooth is not supported

esp_err_t **esp_sleep_enable_wifi_wakeup**(void)

Enable wakeup by WiFi MAC.

Returns

- ESP_OK on success

esp_err_t **esp_sleep_disable_wifi_wakeup**(void)

Disable wakeup by WiFi MAC.

Returns

- ESP_OK on success

uint64_t **esp_sleep_get_ext1_wakeup_status**(void)

Get the bit mask of GPIOs which caused wakeup (ext1)

If wakeup was caused by another source, this function will return 0.

Returns bit mask, if GPIOn caused wakeup, BIT(n) will be set

esp_err_t **esp_sleep_pd_config**(*esp_sleep_pd_domain_t* domain, *esp_sleep_pd_option_t* option)

Set power down mode for an RTC power domain in sleep mode.

If not set set using this API, all power domains default to ESP_PD_OPTION_AUTO.

Parameters

- **domain** –power domain to configure
- **option** –power down option (ESP_PD_OPTION_OFF, ESP_PD_OPTION_ON, or ESP_PD_OPTION_AUTO)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if either of the arguments is out of range

void **esp_deep_sleep_start**(void)

Enter deep sleep with the configured wakeup options.

This function does not return.

esp_err_t **esp_light_sleep_start** (void)

Enter light sleep with the configured wakeup options.

Returns

- ESP_OK on success (returned after wakeup)
- ESP_ERR_SLEEP_REJECT sleep request is rejected(wakeup source set before the sleep request)
- ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION after deducting the sleep flow overhead, the final sleep duration is too short to cover the minimum sleep duration of the chip, when rtc timer wakeup source enabled

void **esp_deep_sleep** (uint64_t time_in_us)

Enter deep-sleep mode.

The device will automatically wake up after the deep-sleep time Upon waking up, the device calls deep sleep wake stub, and then proceeds to load application.

Call to this function is equivalent to a call to esp_deep_sleep_enable_timer_wakeup followed by a call to esp_deep_sleep_start.

esp_deep_sleep does not shut down WiFi, BT, and higher level protocol connections gracefully. Make sure relevant WiFi and BT stack functions are called to close any connections and deinitialize the peripherals. These include:

- esp_bluedroid_disable
- esp_bt_controller_disable
- esp_wifi_stop

This function does not return.

Note: The device will wake up immediately if the deep-sleep time is set to 0

Parameters **time_in_us** – deep-sleep time, unit: microsecond

esp_sleep_wakeup_cause_t **esp_sleep_get_wakeup_cause** (void)

Get the wakeup source which caused wakeup from sleep.

Returns cause of wake up from last sleep (deep sleep or light sleep)

void **esp_wake_deep_sleep** (void)

Default stub to run on wake from deep sleep.

Allows for executing code immediately on wake from sleep, before the software bootloader or ESP-IDF app has started up.

This function is weak-linked, so you can implement your own version to run code immediately when the chip wakes from sleep.

See docs/deep-sleep-stub.rst for details.

void **esp_set_deep_sleep_wake_stub** (*esp_deep_sleep_wake_stub_fn_t* new_stub)

Install a new stub at runtime to run on wake from deep sleep.

If implementing esp_wake_deep_sleep() then it is not necessary to call this function.

However, it is possible to call this function to substitute a different deep sleep stub. Any function used as a deep sleep stub must be marked RTC_IRAM_ATTR, and must obey the same rules given for esp_wake_deep_sleep().

esp_deep_sleep_wake_stub_fn_t **esp_get_deep_sleep_wake_stub** (void)

Get current wake from deep sleep stub.

Returns Return current wake from deep sleep stub, or NULL if no stub is installed.

void **esp_default_wake_deep_sleep**(void)

The default esp-idf-provided esp_wake_deep_sleep() stub.

See docs/deep-sleep-stub.rst for details.

void **esp_deep_sleep_disable_rom_logging**(void)

Disable logging from the ROM code after deep sleep.

Using LSB of RTC_STORE4.

void **esp_sleep_config_gpio_isolate**(void)

Configure to isolate all GPIO pins in sleep state.

void **esp_sleep_enable_gpio_switch**(bool enable)

Enable or disable GPIO pins status switching between slept status and waked status.

Parameters **enable** – decide whether to switch status or not

Type Definitions

typedef *esp_sleep_source_t* **esp_sleep_wakeup_cause_t**

typedef void (***esp_deep_sleep_wake_stub_fn_t**)(void)

Function type for stub to run on wake from sleep.

Enumerations

enum **esp_sleep_ext1_wakeup_mode_t**

Logic function used for EXT1 wakeup mode.

Values:

enumerator **ESP_EXT1_WAKEUP_ALL_LOW**

Wake the chip when all selected GPIOs go low.

enumerator **ESP_EXT1_WAKEUP_ANY_HIGH**

Wake the chip when any of the selected GPIOs go high.

enum **esp_sleep_pd_domain_t**

Power domains which can be powered down in sleep mode.

Values:

enumerator **ESP_PD_DOMAIN_RTC_PERIPH**

RTC IO, sensors and ULP co-processor.

enumerator **ESP_PD_DOMAIN_RTC_SLOW_MEM**

RTC slow memory.

enumerator **ESP_PD_DOMAIN_RTC_FAST_MEM**

RTC fast memory.

enumerator **ESP_PD_DOMAIN_XTAL**

XTAL oscillator.

enumerator **ESP_PD_DOMAIN_RTC8M**

Internal 8M oscillator.

enumerator **ESP_PD_DOMAIN_VDDSDIO**

VDD_SDIO.

enumerator **ESP_PD_DOMAIN_MAX**

Number of domains.

enum **esp_sleep_pd_option_t**

Power down options.

Values:

enumerator **ESP_PD_OPTION_OFF**

Power down the power domain in sleep mode.

enumerator **ESP_PD_OPTION_ON**

Keep power domain enabled during sleep mode.

enumerator **ESP_PD_OPTION_AUTO**

Keep power domain enabled in sleep mode, if it is needed by one of the wakeup options. Otherwise power it down.

enum **esp_sleep_source_t**

Sleep wakeup cause.

Values:

enumerator **ESP_SLEEP_WAKEUP_UNDEFINED**

In case of deep sleep, reset was not caused by exit from deep sleep.

enumerator **ESP_SLEEP_WAKEUP_ALL**

Not a wakeup cause, used to disable all wakeup sources with esp_sleep_disable_wakeup_source.

enumerator **ESP_SLEEP_WAKEUP_EXT0**

Wakeup caused by external signal using RTC_IO.

enumerator **ESP_SLEEP_WAKEUP_EXT1**

Wakeup caused by external signal using RTC_CNTL.

enumerator **ESP_SLEEP_WAKEUP_TIMER**

Wakeup caused by timer.

enumerator **ESP_SLEEP_WAKEUP_TOUCHPAD**

Wakeup caused by touchpad.

enumerator **ESP_SLEEP_WAKEUP_ULP**

Wakeup caused by ULP program.

enumerator **ESP_SLEEP_WAKEUP_GPIO**

Wakeup caused by GPIO (light sleep only on ESP32, S2 and S3)

enumerator **ESP_SLEEP_WAKEUP_UART**

Wakeup caused by UART (light sleep only)

enumerator **ESP_SLEEP_WAKEUP_WIFI**

Wakeup caused by WIFI (light sleep only)

enumerator **ESP_SLEEP_WAKEUP_COCPU**

Wakeup caused by COCPU int.

enumerator **ESP_SLEEP_WAKEUP_COCPU_TRAP_TRIG**

Wakeup caused by COCPU crash.

enumerator **ESP_SLEEP_WAKEUP_BT**

Wakeup caused by BT (light sleep only)

enum **[anonymous]**

Values:

enumerator **ESP_ERR_SLEEP_REJECT**

enumerator **ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION**

2.10.27 SoC Capabilities

This section lists definitions of the ESP32 ' s SoC hardware capabilities. These definitions are commonly used in IDF to control which hardware dependent features are supported and thus compiled into the binary.

Note: These defines are currently not considered to be part of the public API, and may be changed at any time.

API Reference

Header File

- components/soc/esp32/include/soc/soc_caps.h

Macros

SOC_CAPS_ECO_VER_MAX

SOC_ADC_SUPPORTED

SOC_DAC_SUPPORTED

SOC_MCPWM_SUPPORTED

SOC_SDMMC_HOST_SUPPORTED

SOC_BT_SUPPORTED

SOC_PCNT_SUPPORTED

SOC_WIFI_SUPPORTED

SOC_SDIO_SLAVE_SUPPORTED

SOC_TWAI_SUPPORTED

SOC_EMAC_SUPPORTED

SOC_ULP_SUPPORTED

SOC_CCOMP_TIMER_SUPPORTED

SOC_RTC_FAST_MEM_SUPPORTED

SOC_RTC_SLOW_MEM_SUPPORTED

SOC_RTC_MEM_SUPPORTED

SOC_I2S_SUPPORTED

SOC_RMT_SUPPORTED

SOC_SDM_SUPPORTED

SOC_SUPPORT_COEXISTENCE

SOC_AES_SUPPORTED

SOC_MPI_SUPPORTED

SOC_SHA_SUPPORTED

SOC_FLASH_ENC_SUPPORTED

SOC_SECURE_BOOT_SUPPORTED

SOC_TOUCH_SENSOR_SUPPORTED

SOC_DPORT_WORKAROUND

SOC_DPORT_WORKAROUND_DIS_INTERRUPT_LVL

SOC_XTAL_SUPPORT_26M

SOC_XTAL_SUPPORT_40M

SOC_XTAL_SUPPORT_AUTO_DETECT

SOC_ADC_RTC_CTRL_SUPPORTED
< SAR ADC Module

SOC_ADC_DIG_CTRL_SUPPORTED

SOC_ADC_DMA_SUPPORTED

SOC_ADC_DIG_SUPPORTED_UNIT(UNIT)

SOC_ADC_PERIPH_NUM

SOC_ADC_CHANNEL_NUM(PERIPH_NUM)

SOC_ADC_MAX_CHANNEL_NUM

SOC_ADC_ATTEN_NUM
Digital

SOC_ADC_DIGI_CONTROLLER_NUM

SOC_ADC_PATT_LEN_MAX

SOC_ADC_DIGI_MIN_BITWIDTH

SOC_ADC_DIGI_MAX_BITWIDTH

SOC_ADC_DIGI_RESULT_BYTES

SOC_ADC_DIGI_DATA_BYTES_PER_CONV

SOC_ADC_SAMPLE_FREQ_THRES_HIGH

SOC_ADC_SAMPLE_FREQ_THRES_LOW
RTC

SOC_ADC_RTC_MIN_BITWIDTH

SOC_ADC_RTC_MAX_BITWIDTH

SOC_RTC_SLOW_CLOCK_SUPPORT_8MD256

SOC_SHARED_IDCACHE_SUPPORTED

SOC_IDCACHE_PER_CORE

SOC_MMU_LINEAR_ADDRESS_REGION_NUM

SOC_CPU_CORES_NUM

SOC_CPU_INTR_NUM

SOC_CPU_HAS_FPU

SOC_CPU_BREAKPOINTS_NUM

SOC_CPU_WATCHPOINTS_NUM

SOC_CPU_WATCHPOINT_MAX_REGION_SIZE

SOC_DAC_PERIPH_NUM

SOC_DAC_RESOLUTION

SOC_GPIO_PORT

SOC_GPIO_PIN_COUNT

SOC_GPIO_VALID_GPIO_MASK

SOC_GPIO_VALID_OUTPUT_GPIO_MASK

SOC_GPIO_VALID_DIGITAL_IO_PAD_MASK

SOC_I2C_NUM

SOC_I2C_FIFO_LEN
I2C hardware FIFO depth

SOC_I2C_SUPPORT_SLAVE

SOC_I2C_SUPPORT_APB

SOC_CLK_APLL_SUPPORTED

SOC_APLL_MULTIPLIER_OUT_MIN_HZ

SOC_APLL_MULTIPLIER_OUT_MAX_HZ

SOC_APLL_MIN_HZ

SOC_APLL_MAX_HZ

SOC_I2S_NUM

SOC_I2S_HW_VERSION_1

SOC_I2S_SUPPORTS_APLL

SOC_I2S_SUPPORTS_PDM

SOC_I2S_SUPPORTS_PDM_TX

SOC_I2S_SUPPORTS_PDM_RX

SOC_I2S_SUPPORTS_ADC_DAC

SOC_I2S_SUPPORTS_ADC

SOC_I2S_SUPPORTS_DAC

SOC_I2S_SUPPORTS_LCD_CAMERA

SOC_I2S_TRANS_SIZE_ALIGN_WORD

SOC_I2S_LCD_I80_VARIANT

SOC_LCD_I80_SUPPORTED
Intel 8080 LCD is supported

SOC_LCD_I80_BUSES
Both I2S0/1 have LCD mode

SOC_LCD_I80_BUS_WIDTH
Intel 8080 bus width

SOC_LEDC_HAS_TIMER_SPECIFIC_MUX

SOC_LEDC_SUPPORT_APB_CLOCK

SOC_LEDC_SUPPORT_REF_TICK

SOC_LEDC_SUPPORT_HS_MODE

SOC_LEDC_CHANNEL_NUM

SOC_LEDC_TIMER_BIT_WIDE_NUM

SOC_MCPWM_GROUPS

2 MCPWM groups on the chip (i.e., the number of independent MCPWM peripherals)

SOC_MCPWM_TIMERS_PER_GROUP

The number of timers that each group has.

SOC_MCPWM_OPERATORS_PER_GROUP

The number of operators that each group has.

SOC_MCPWM_COMPARATORS_PER_OPERATOR

The number of comparators that each operator has.

SOC_MCPWM_GENERATORS_PER_OPERATOR

The number of generators that each operator has.

SOC_MCPWM_TRIGGERS_PER_OPERATOR

The number of triggers that each operator has.

SOC_MCPWM_GPIO_FAULTS_PER_GROUP

The number of GPIO fault signals that each group has.

SOC_MCPWM_CAPTURE_TIMERS_PER_GROUP

The number of capture timers that each group has.

SOC_MCPWM_CAPTURE_CHANNELS_PER_TIMER

The number of capture channels that each capture timer has.

SOC_MCPWM_GPIO_SYNCHROS_PER_GROUP

The number of GPIO synchros that each group has.

SOC_MPU_CONFIGURABLE_REGIONS_SUPPORTED

SOC_MPU_MIN_REGION_SIZE

SOC_MPU_REGIONS_MAX_NUM

SOC_MPU_REGION_RO_SUPPORTED

SOC_MPU_REGION_WO_SUPPORTED

SOC_PCNT_GROUPS

SOC_PCNT_UNITS_PER_GROUP

SOC_PCNT_CHANNELS_PER_UNIT

SOC_PCNT_THRES_POINT_PER_UNIT

SOC_RMT_GROUPS
One RMT group

SOC_RMT_TX_CANDIDATES_PER_GROUP
Number of channels that capable of Transmit in each group

SOC_RMT_RX_CANDIDATES_PER_GROUP
Number of channels that capable of Receive in each group

SOC_RMT_CHANNELS_PER_GROUP
Total 8 channels

SOC_RMT_MEM_WORDS_PER_CHANNEL
Each channel owns 64 words memory

SOC_RMT_SUPPORT_REF_TICK
Support set REF_TICK as the RMT clock source

SOC_RMT_SUPPORT_APB
Support set APB as the RMT clock source

SOC_RMT_CHANNEL_CLK_INDEPENDENT
Can select different source clock for each channel

SOC_RTCIO_PIN_COUNT

SOC_RTCIO_INPUT_OUTPUT_SUPPORTED

SOC_RTCIO_HOLD_SUPPORTED

SOC_RTCIO_WAKE_SUPPORTED

SOC_SDM_GROUPS

SOC_SDM_CHANNELS_PER_GROUP

SOC_SPI_HD_BOTH_INOUT_SUPPORTED

SOC_SPI_AS_CS_SUPPORTED

SOC_SPI_PERIPH_NUM

SOC_SPI_DMA_CHAN_NUM

SOC_SPI_PERIPH_CS_NUM(i)

SOC_SPI_MAX_CS_NUM

SOC_SPI_MAXIMUM_BUFFER_SIZE

SOC_SPI_MAX_PRE_DIVIDER

SOC_MEMSPI_SRC_FREQ_80M_SUPPORTED

SOC_MEMSPI_SRC_FREQ_40M_SUPPORTED

SOC_MEMSPI_SRC_FREQ_26M_SUPPORTED

SOC_MEMSPI_SRC_FREQ_20M_SUPPORTED

SOC_SPI_PERIPH_SUPPORT_MULTILINE_MODE(spi_host)

SOC_TIMER_GROUPS

SOC_TIMER_GROUP_TIMERS_PER_GROUP

SOC_TIMER_GROUP_COUNTER_BIT_WIDTH

SOC_TIMER_GROUP_TOTAL_TIMERS

SOC_TIMER_GROUP_SUPPORT_APB

SOC_TOUCH_VERSION_1

Hardware version of touch sensor

SOC_TOUCH_SENSOR_NUM

SOC_TOUCH_PAD_MEASURE_WAIT_MAX

The timer frequency is 8Mhz, the max value is 0xff

SOC_TOUCH_PAD_THRESHOLD_MAX

If set touch threshold max value, The touch sensor can’ t be in touched status

SOC_TWAI_BRP_MIN

SOC_TWAI_BRP_MAX

SOC_TWAI_SUPPORT_MULTI_ADDRESS_LAYOUT

SOC_UART_NUM

SOC_UART_SUPPORT_APB_CLK
Support APB as the clock source

SOC_UART_SUPPORT_REF_TICK
Support REF_TICK as the clock source

SOC_UART_FIFO_LEN
The UART hardware FIFO length

SOC_UART_BITRATE_MAX
Max bit rate supported by UART

SOC_SPIRAM_SUPPORTED

SOC_SPI_MEM_SUPPORT_CONFIG_GPIO_BY_EFUSE

SOC_SHA_SUPPORT_PARALLEL_ENG

SOC_SHA_SUPPORT_SHA1

SOC_SHA_SUPPORT_SHA256

SOC_SHA_SUPPORT_SHA384

SOC_SHA_SUPPORT_SHA512

SOC_RSA_MAX_BIT_LEN

SOC_AES_SUPPORT_AES_128

SOC_AES_SUPPORT_AES_192

SOC_AES_SUPPORT_AES_256

SOC_SECURE_BOOT_V1

SOC_EFUSE_SECURE_BOOT_KEY_DIGESTS

SOC_FLASH_ENCRYPTED_XTS_AES_BLOCK_MAX

SOC_PHY_DIG_REGS_MEM_SIZE

SOC_PM_SUPPORT_EXT_WAKEUP

SOC_PM_SUPPORT_TOUCH_SENSOR_WAKEUP
Supports waking up from touch pad trigger

SOC_PM_SUPPORT_RTC_PERIPH_PD

SOC_PM_SUPPORT_RTC_FAST_MEM_PD

SOC_PM_SUPPORT_RTC_SLOW_MEM_PD

SOC_PM_SUPPORT_MODEM_PD
Modem here includes wifi and btdm

SOC_SDMMC_USE_IOMUX

SOC_SDMMC_NUM_SLOTS

SOC_WIFI_HW_TSF
Hardware TSF is not supported

SOC_WIFI_FTM_SUPPORT
FTM is not supported

SOC_WIFI_GCMP_SUPPORT
GCMP is not supported(GCMP128 and GCMP256)

SOC_WIFI_WAPI_SUPPORT
Support WAPI

SOC_WIFI_CSI_SUPPORT
Support CSI

SOC_WIFI_MESH_SUPPORT
Support WIFI MESH

SOC_BLE_SUPPORTED
Support Bluetooth Low Energy hardware

SOC_BLE_MESH_SUPPORTED
Support BLE MESH

SOC_BT_CLASSIC_SUPPORTED
Support Bluetooth Classic hardware

SOC_BLE_DEVICE_PRIVACY_SUPPORTED
Support BLE device privacy mode

2.10.28 System Time

Overview

ESP32 uses two hardware timers for the purpose of keeping system time. System time can be kept by using either one or both of the hardware timers depending on the application's purpose and accuracy requirements for system time. The two hardware timers are:

- **RTC timer**: This timer allows time keeping in various sleep modes, and can also persist time keeping across any resets (with the exception of power-on resets which reset the RTC timer). The frequency deviation depends on the *RTC Timer Clock Sources* and affects the accuracy only in sleep modes, in which case the time will be measured at 6.6667 µs resolution.
- **High-resolution timer**: This timer is not available in sleep modes and will not persist over a reset, but has greater accuracy. The timer uses the APB_CLK clock source (typically 80 MHz), which has a frequency deviation of less than ±10 ppm. Time will be measured at 1 µs resolution.

The possible combinations of hardware timers used to keep system time are listed below:

- RTC and high-resolution timer (default)
- RTC
- High-resolution timer
- None

It is recommended that users stick to the default option as it provides the highest accuracy. However, users can also select a different setting via the *CONFIG_NEWLIB_TIME_SYSCALL* configuration option.

RTC Timer Clock Sources

The RTC timer has the following clock sources:

- `Internal 150 kHz RC oscillator` (default): Features the lowest Deep-sleep current consumption and no dependence on any external components. However, the frequency stability of this clock source is affected by temperature fluctuations, so time may drift in both Deep-sleep and Light-sleep modes.
- `External 32 kHz crystal`: Requires a 32 kHz crystal to be connected to the 32K_XP and 32K_XN pins. This source provides a better frequency stability at the expense of a slightly higher (by 1 µA) Deep-sleep current consumption.
- `External 32 kHz oscillator at 32K_XN pin`: Allows using 32 kHz clock generated by an external circuit. The external clock signal must be connected to the 32K_XN pin. The amplitude should be less than 1.2 V for sine wave signal and less than 1 V for square wave signal. Common mode voltage should be in the range of 0.1 < Vcm < 0.5xVamp, where Vamp stands for signal amplitude. In this case, the 32K_XN pin cannot be used as a GPIO pin.
- `Internal 8.5 MHz oscillator, divided by 256 (~33 kHz)`: Provides better frequency stability than the `Internal 150 kHz RC oscillator` at the expense of a higher (by 5 µA) Deep-sleep current consumption. It also does not require external components.

The choice depends on your requirements for system time accuracy and power consumption in sleep modes. To modify the RTC clock source, set *CONFIG_RTC_CLK_SRC* in project configuration.

More details about the wiring requirements for the external crystal or external oscillator, please refer to ESP32 Hardware Design Guidelines.

Get Current Time

To get the current time, use the POSIX function `gettimeofday()`. Additionally, you can use the following standard C library functions to obtain time and manipulate it:

```
gettimeofday
time
asctime
clock
```

(continues on next page)

(continued from previous page)

```
ctime
difftime
gmtime
localtime
mktime
strftime
adjtime*
```

To stop smooth time adjustment and update the current time immediately, use the POSIX function `settimeofday()`.

If you need to obtain time with one second resolution, use the following code snippet:

```
time_t now;
char strftime_buf[64];
struct tm timeinfo;

time(&now);
// Set timezone to China Standard Time
setenv("TZ", "CST-8", 1);
tzset();

localtime_r(&now, &timeinfo);
strftime(strftime_buf, sizeof(strftime_buf), "%c", &timeinfo);
ESP_LOGI(TAG, "The current date/time in Shanghai is: %s", strftime_buf);
```

If you need to obtain time with one microsecond resolution, use the code snippet below:

```
struct timeval tv_now;
gettimeofday(&tv_now, NULL);
int64_t time_us = (int64_t)tv_now.tv_sec * 1000000L + (int64_t)tv_now.tv_usec;
```

SNTP Time Synchronization

To set the current time, you can use the POSIX functions `settimeofday()` and `adjtime()`. They are used internally in the lwIP SNTP library to set current time when a response from the NTP server is received. These functions can also be used separately from the lwIP SNTP library.

The function to use inside the lwIP SNTP library depends on the sync mode for system time. Use the function *`sntp_set_sync_mode()`* to set one of the following sync modes:

- *`SNTP_SYNC_MODE_IMMED`* (default): Updates system time immediately upon receiving a response from the SNTP server after using `settimeofday()`.
- *`SNTP_SYNC_MODE_SMOOTH`*: Updates time smoothly by gradually reducing time error using the function `adjtime()`. If the difference between the SNTP response time and system time is more than 35 minutes, update system time immediately by using `settimeofday()`.

The lwIP SNTP library has API functions for setting a callback function for a certain event. You might need the following functions:

- *`sntp_set_time_sync_notification_cb()`*: Can be used to set a callback function that will notify of the time synchronization process.
- *`sntp_get_sync_status()`* and *`sntp_set_sync_status()`*: Can be used to get/set time synchronization status.

To start synchronization via SNTP, just call the following three functions:

```
esp_sntp_setoperatingmode(ESP_SNTP_OPMODE_POLL);
esp_sntp_setservername(0, "pool.ntp.org");
esp_sntp_init();
```

An application with this initialization code will periodically synchronize the time. The time synchronization period is determined by *CONFIG_LWIP_SNTP_UPDATE_DELAY* (the default value is one hour). To modify the variable, set *CONFIG_LWIP_SNTP_UPDATE_DELAY* in project configuration.

A code example that demonstrates the implementation of time synchronization based on the lwIP SNTP library is provided in the protocols/sntp directory.

Timezones

To set the local timezone, use the following POSIX functions:

1. Call `setenv()` to set the `TZ` environment variable to the correct value based on the device location. The format of the time string is the same as described in the GNU libc documentation (although the implementation is different).
2. Call `tzset()` to update C library runtime data for the new timezone.

Once these steps are completed, call the standard C library function `localtime()`, and it will return the correct local time taking into account the timezone offset and daylight saving time.

Year 2036 and 2038 Overflow Issues

SNTP/NTP 2036 Overflow SNTP/NTP timestamps are represented as 64-bit unsigned fixed point numbers, where the first 32 bits represent the integer part, and the last 32 bits represent the fractional part. The 64-bit unsigned fixed point number represents the number of seconds since 00:00 on 1st of January 1900, thus SNTP/NTP times will overflow in the year 2036.

To address this issue, lifetime of the SNTP/NTP timestamps has been extended by convention by using the MSB (bit 0 by convention) of the integer part to indicate time ranges between years 1968 to 2104 (see RFC2030 for more details). This convention is implemented in lwIP library SNTP module. Therefore SNTP-related functions in ESP-IDF are future-proof until year 2104.

Unix Time 2038 Overflow Unix time (type `time_t`) was previously represented as a 32-bit signed integer, leading to an overflow in year 2038 (i.e., Y2K38 issue). To address the Y2K38 issue, ESP-IDF uses a 64-bit signed integer to represent `time_t` starting from release v5.0, thus deferring `time_t` overflow for another 292 billion years.

API Reference

Header File

- components/lwip/include/apps/esp_sntp.h

Functions

void **sntp_sync_time** (struct timeval *tv)

This function updates the system time.

This is a weak-linked function. It is possible to replace all SNTP update functionality by placing a sntp_sync_time() function in the app firmware source. If the default implementation is used, calling sntp_set_sync_mode() allows the time synchronization mode to be changed to instant or smooth. If a callback function is registered via sntp_set_time_sync_notification_cb(), it will be called following time synchronization.

Parameters **tv** –Time received from SNTP server.

void **sntp_set_sync_mode** (*sntp_sync_mode_t* sync_mode)

Set the sync mode.

Modes allowed: SNTP_SYNC_MODE_IMMED and SNTP_SYNC_MODE_SMOOTH.

Parameters **sync_mode** –Sync mode.

sntp_sync_mode_t **sntp_get_sync_mode** (void)

Get set sync mode.

Returns SNTP_SYNC_MODE_IMMED: Update time immediately. SNTP_SYNC_MODE_SMOOTH: Smooth time updating.

sntp_sync_status_t **sntp_get_sync_status** (void)

Get status of time sync.

After the update is completed, the status will be returned as SNTP_SYNC_STATUS_COMPLETED. After that, the status will be reset to SNTP_SYNC_STATUS_RESET. If the update operation is not completed yet, the status will be SNTP_SYNC_STATUS_RESET. If a smooth mode was chosen and the synchronization is still continuing (adjtime works), then it will be SNTP_SYNC_STATUS_IN_PROGRESS.

Returns SNTP_SYNC_STATUS_RESET: Reset status. SNTP_SYNC_STATUS_COMPLETED: Time is synchronized. SNTP_SYNC_STATUS_IN_PROGRESS: Smooth time sync in progress.

void **sntp_set_sync_status** (*sntp_sync_status_t* sync_status)

Set status of time sync.

Parameters **sync_status** –status of time sync (see sntp_sync_status_t)

void **sntp_set_time_sync_notification_cb** (*sntp_sync_time_cb_t* callback)

Set a callback function for time synchronization notification.

Parameters **callback** –a callback function

void **sntp_set_sync_interval** (uint32_t interval_ms)

Set the sync interval of SNTP operation.

Note: SNTPv4 RFC 4330 enforces a minimum sync interval of 15 seconds. This sync interval will be used in the next attempt update time throught SNTP. To apply the new sync interval call the sntp_restart() function, otherwise, it will be applied after the last interval expired.

Parameters **interval_ms** –The sync interval in ms. It cannot be lower than 15 seconds, otherwise 15 seconds will be set.

uint32_t **sntp_get_sync_interval** (void)

Get the sync interval of SNTP operation.

Returns the sync interval

bool **sntp_restart** (void)

Restart SNTP.

Returns True - Restart False - SNTP was not initialized yet

void **esp_sntp_setoperatingmode** (*esp_sntp_operatingmode_t* operating_mode)

Sets SNTP operating mode. The mode has to be set before init.

Parameters **operating_mode** –Desired operating mode

void **esp_sntp_init** (void)

Init and start SNTP service.

void **esp_sntp_stop** (void)

Stops SNTP service.

void **esp_sntp_setserver** (u8_t idx, const ip_addr_t *addr)

Sets SNTP server address.

Parameters

- **idx** –Index of the server
- **addr** –IP address of the server

void **esp_sntp_setservername**(u8_t idx, const char *server)

Sets SNTP hostname.

Parameters
- **idx** –Index of the server
- **server** –Name of the server

const char ***esp_sntp_getservername**(u8_t idx)

Gets SNTP server name.

Parameters **idx** –Index of the server
Returns Name of the server

const ip_addr_t ***esp_sntp_getserver**(u8_t idx)

Get SNTP server IP.

Parameters **idx** –Index of the server
Returns IP address of the server

bool **esp_sntp_enabled**(void)

Checks if sntp is enabled.

Returns true if sntp module is enabled

Macros

esp_sntp_sync_time

Aliases for esp_sntp prefixed API (inherently thread safe)

esp_sntp_set_sync_mode

esp_sntp_get_sync_mode

esp_sntp_get_sync_status

esp_sntp_set_sync_status

esp_sntp_set_time_sync_notification_cb

esp_sntp_set_sync_interval

esp_sntp_get_sync_interval

esp_sntp_restart

Type Definitions

typedef void (***sntp_sync_time_cb_t**)(struct timeval *tv)

SNTP callback function for notifying about time sync event.

Param tv Time received from SNTP server.

Enumerations

enum **sntp_sync_mode_t**

SNTP time update mode.

Values:

enumerator **SNTP_SYNC_MODE_IMMED**

Update system time immediately when receiving a response from the SNTP server.

enumerator **SNTP_SYNC_MODE_SMOOTH**

Smooth time updating. Time error is gradually reduced using adjtime function. If the difference between SNTP response time and system time is large (more than 35 minutes) then update immediately.

enum **sntp_sync_status_t**

SNTP sync status.

Values:

enumerator **SNTP_SYNC_STATUS_RESET**

enumerator **SNTP_SYNC_STATUS_COMPLETED**

enumerator **SNTP_SYNC_STATUS_IN_PROGRESS**

enum **esp_sntp_operatingmode_t**

SNTP operating modes per lwip SNTP module.

Values:

enumerator **ESP_SNTP_OPMODE_POLL**

enumerator **ESP_SNTP_OPMODE_LISTENONLY**

2.10.29 The himem allocation API

Overview

The ESP32 can access external SPI RAM transparently, so you can use it as normal memory in your program code. However, because the address space for external memory is limited in size, only the first 4MiB can be used as such. Access to the remaining memory is still possible, however this needs to go through a bankswitching scheme controlled by the himem API.

Specifically, what is implemented by the himem API is a bankswitching scheme. Hardware-wise, the 4MiB region for external SPI RAM is mapped into the CPU address space by a MMU, which maps a configurable 32K bank/page of external SPI RAM into each of the 32K pages in the 4MiB region accessed by the CPU. For external memories that are <=4MiB, this MMU is configured to unity mapping, effectively mapping each CPU address 1-to-1 to the external SPI RAM address.

In order to use the himem API, you have to enable it in the menuconfig using *CONFIG_SPIRAM_BANKSWITCH_ENABLE*, as well as set the amount of banks reserved for this in *CONFIG_SPIRAM_BANKSWITCH_RESERVE*. This decreases the amount of external memory allocated by functions like `malloc()`, but it allows you to use the himem api to map any of the remaining memory into the reserved banks.

The himem API is more-or-less an abstraction of the bankswitching scheme: it allows you to claim one or more banks of address space (called 'regions' in the API) as well as one or more of banks of memory to map into the ranges.

Example

An example doing a simple memory test of the high memory range is available in esp-idf: system/himem

API Reference

Header File

- components/esp_psram/include/esp32/himem.h

Functions

esp_err_t **esp_himem_alloc**(size_t size, *esp_himem_handle_t* *handle_out)

Allocate a block in high memory.

Parameters
- **size** –Size of the to-be-allocated block, in bytes. Note that this needs to be a multiple of the external RAM mmu block size (32K).
- **handle_out** –**[out]** Handle to be returned

Returns - ESP_OK if succesful
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_SIZE if size is not a multiple of 32K

esp_err_t **esp_himem_alloc_map_range**(size_t size, *esp_himem_rangehandle_t* *handle_out)

Allocate a memory region to map blocks into.

This allocates a contiguous CPU memory region that can be used to map blocks of physical memory into.

Parameters
- **size** –Size of the range to be allocated. Note this needs to be a multiple of the external RAM mmu block size (32K).
- **handle_out** –**[out]** Handle to be returned

Returns - ESP_OK if succesful
- ESP_ERR_NO_MEM if out of memory or address space
- ESP_ERR_INVALID_SIZE if size is not a multiple of 32K

esp_err_t **esp_himem_map**(*esp_himem_handle_t* handle, *esp_himem_rangehandle_t* range, size_t ram_offset, size_t range_offset, size_t len, int flags, void **out_ptr)

Map a block of high memory into the CPUs address space.

This effectively makes the block available for read/write operations.

Note: The region to be mapped needs to have offsets and sizes that are aligned to the SPI RAM MMU block size (32K)

Parameters
- **handle** –Handle to the block of memory, as given by esp_himem_alloc
- **range** –Range handle to map the memory in
- **ram_offset** –Offset into the block of physical memory of the block to map
- **range_offset** –Offset into the address range where the block will be mapped
- **len** –Length of region to map
- **flags** –One of ESP_HIMEM_MAPFLAG_*
- **out_ptr** –**[out]** Pointer to variable to store resulting memory pointer in

Returns - ESP_OK if the memory could be mapped
- ESP_ERR_INVALID_ARG if offset, range or len aren't MMU-block-aligned (32K)
- ESP_ERR_INVALID_SIZE if the offsets/lengths don't fit in the allocated memory or range
- ESP_ERR_INVALID_STATE if a block in the selected ram offset/length is already mapped, or if a block in the selected range offset/length already has a mapping.

esp_err_t **esp_himem_free**(*esp_himem_handle_t* handle)

Free a block of physical memory.

This clears out the associated handle making the memory available for re-allocation again. This will only succeed if none of the memory blocks currently have a mapping.

Parameters **handle** –Handle to the block of memory, as given by esp_himem_alloc
Returns - ESP_OK if the memory is succesfully freed
- ESP_ERR_INVALID_ARG if the handle still is (partially) mapped

esp_err_t **esp_himem_free_map_range**(*esp_himem_rangehandle_t* handle)

Free a mapping range.

This clears out the associated handle making the range available for re-allocation again. This will only succeed if none of the range blocks currently are used for a mapping.

Parameters **handle** –Handle to the range block, as given by esp_himem_alloc_map_range
Returns - ESP_OK if the memory is succesfully freed
- ESP_ERR_INVALID_ARG if the handle still is (partially) mapped to

esp_err_t **esp_himem_unmap**(*esp_himem_rangehandle_t* range, void *ptr, size_t len)

Unmap a region.

Parameters
- **range** –Range handle
- **ptr** –Pointer returned by esp_himem_map
- **len** –Length of the block to be unmapped. Must be aligned to the SPI RAM MMU blocksize (32K)

Returns - ESP_OK if the memory is succesfully unmapped,
- ESP_ERR_INVALID_ARG if ptr or len are invalid.

size_t **esp_himem_get_phys_size**(void)

Get total amount of memory under control of himem API.

Returns Amount of memory, in bytes

size_t **esp_himem_get_free_size**(void)

Get free amount of memory under control of himem API.

Returns Amount of free memory, in bytes

size_t **esp_himem_reserved_area_size**(void)

Get amount of SPI memory address space needed for bankswitching.

Note: This is also weakly defined in esp32/spiram.c and returns 0 there, so if no other function in this file is used, no memory is reserved.

Returns Amount of reserved area, in bytes

Macros

ESP_HIMEM_BLKSZ

ESP_HIMEM_MAPFLAG_RO

Indicates that a mapping will only be read from. Note that this is unused for now.

Type Definitions

typedef struct esp_himem_ramdata_t ***esp_himem_handle_t**

typedef struct esp_himem_rangedata_t ***esp_himem_rangehandle_t**

2.10.30 ULP Coprocessor programming

The Ultra Low Power (ULP) coprocessor is a simple finite state machine (FSM) which is designed to perform measurements using the ADC, temperature sensor, and external I2C sensors, while the main processors are in deep sleep mode. The ULP coprocessor can access the RTC_SLOW_MEM memory region, and registers in the RTC_CNTL, RTC_IO, and SARADC peripherals. The ULP coprocessor uses fixed-width 32-bit instructions, 32-bit memory addressing, and has 4 general-purpose 16-bit registers. This coprocessor is referred to as *ULP FSM* in ESP-IDF.

Installing the Toolchain

The ULP FSM coprocessor code is written in assembly and compiled using the binutils-esp32ulp toolchain.

If you have already set up ESP-IDF with CMake build system according to the *Getting Started Guide*, then the ULP FSM toolchain will already be installed.

Programming ULP FSM

The ULP FSM can be programmed using the supported instruction set. Alternatively, the ULP FSM coprocessor can also be programmed using C Macros on the main CPU. Theses two methods are described in the following section:

ESP32 ULP coprocessor instruction set This document provides details about the instructions used by ESP32 ULP FSM coprocessor assembler.

ULP FSM coprocessor has 4 16-bit general purpose registers, labeled R0, R1, R2, R3. It also has an 8-bit counter register (stage_cnt) which can be used to implement loops. Stage count register is accessed using special instructions.

ULP coprocessor can access 8k bytes of RTC_SLOW_MEM memory region. Memory is addressed in 32-bit word units. It can also access peripheral registers in RTC_CNTL, RTC_IO, and SENS peripherals.

All instructions are 32-bit. Jump instructions, ALU instructions, peripheral register and memory access instructions are executed in 1 cycle. Instructions which work with peripherals (TSENS, ADC, I2C) take variable number of cycles, depending on peripheral operation.

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily. This is true both for register names and instruction names.

Note about addressing ESP32 ULP FSM coprocessor's `JUMP`, `ST`, `LD` family of instructions expect the address argument to be expressed in the following way depending on the type of address argument used:

- When the address argument is presented as a label then the instruction expects the address to be expressed as 32-bit words.
  Consider the following example program:

```
entry:
        NOP
        NOP
        NOP
        NOP
loop:
        MOVE R1, loop
        JUMP R1
```

When this program is assembled and linked, address of label `loop` will be equal to 16 (expressed in bytes). However *JUMP* instruction expects the address stored in register `R1` to be expressed in 32-bit words. To account for this common use case, the assembler will convert the address of label *loop* from bytes to words, when generating the `MOVE` instruction. Hence, the code generated code will be equivalent to:

```
0000    NOP
0004    NOP
0008    NOP
000c    NOP
0010    MOVE R1, 4
0014    JUMP R1
```

- The other case is when the argument of `MOVE` instruction is not a label but a constant. In this case assembler will **use the value as is**, without any conversion:

```
.set        val, 0x10
MOVE        R1, val
```

In this case, value loaded into `R1` will be `0x10`.
However, when an immediate value is used as an offset in `LD` and `ST` instructions, the assembler considers the address argument in bytes and converts it to 32-bit words before executing the instruction:

```
ST R1, R2, 4        // offset = 4 bytes; Mem[R2 + 4 / 4] = R1
```

In this case, the value in R1 is stored at the memory location pointed by [R2 + offset / 4]
Consider the following code:

```
        .global array
array:  .long 0
        .long 0
        .long 0
        .long 0

        MOVE R1, array
        MOVE R2, 0x1234
        ST R2, R1, 0        // write value of R2 into the first array element,
                            // i.e. array[0]

        ST R2, R1, 4        // write value of R2 into the second array element
                            // (4 byte offset), i.e. array[1]

        ADD R1, R1, 2       // this increments address by 2 words (8 bytes)
        ST R2, R1, 0        // write value of R2 into the third array element,
                            // i.e. array[2]
```

Note about instruction execution time ULP coprocessor is clocked from RTC_FAST_CLK, which is normally derived from the internal 8MHz oscillator. Applications which need to know exact ULP clock frequency can calibrate it against the main XTAL clock:

```
#include "soc/rtc.h"

// calibrate 8M/256 clock against XTAL, get 8M/256 clock period
uint32_t rtc_8md256_period = rtc_clk_cal(RTC_CAL_8MD256, 100);
uint32_t rtc_fast_freq_hz = 1000000ULL * (1 << RTC_CLK_CAL_FRACT) * 256 / rtc_
↪8md256_period;
```

ULP coprocessor needs certain number of clock cycles to fetch each instruction, plus certain number of cycles to execute it, depending on the instruction. See description of each instruction below for details on the execution time.

Instruction fetch time is:

- 2 clock cycles —for instructions following ALU and branch instructions.
- 4 clock cycles —in other cases.

Note that when accessing RTC memories and RTC registers, ULP coprocessor has lower priority than the main CPUs. This means that ULP coprocessor execution may be suspended while the main CPUs access same memory region as the ULP.

The detailed description of all instructions is presented below:

NOP - no operation

Syntax **NOP**
Operands None
Cycles 2 cycle to execute, 4 cycles to fetch next instruction
Description No operation is performed. Only the PC is incremented.

Example:

```
1:     NOP
```

ADD - Add to register

Syntax **ADD** *Rdst, Rsrc1, Rsrc2*
ADD *Rdst, Rsrc1, imm*
Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction
Description The instruction adds source register to another source register or to a 16-bit signed value and stores the result in the destination register.

Examples:

```
1:     ADD R1, R2, R3          // R1 = R2 + R3

2:     Add R1, R2, 0x1234      // R1 = R2 + 0x1234

3:     .set value1, 0x03       // constant value1=0x03
       Add R1, R2, value1      // R1 = R2 + value1

4:     .global label           // declaration of variable label
       add R1, R2, label       // R1 = R2 + label
         ...
       label: nop              // definition of variable label
```

SUB - Subtract from register

Syntax **SUB** *Rdst, Rsrc1, Rsrc2*
SUB *Rdst, Rsrc1, imm*
Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction
Description The instruction subtracts the source register from another source register or subtracts a 16-bit signed value from a source register, and stores the result to the destination register.

Examples:

```
1:          SUB R1, R2, R3              // R1 = R2 - R3

2:          sub R1, R2, 0x1234          // R1 = R2 - 0x1234

3:          .set value1, 0x03           // constant value1=0x03
            SUB R1, R2, value1          // R1 = R2 - value1
4:          .global label               // declaration of variable label
            SUB R1, R2, label           // R1 = R2 - label
              ....
  label:    nop                         // definition of variable label
```

AND - Bitwise logical AND of two operands

Syntax **AND** *Rdst, Rsrc1, Rsrc2*
AND *Rdst, Rsrc1, imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does a bitwise logical AND of a source register and another source register or a 16-bit signed value and stores the result to the destination register.

Examples:

```
1:          AND R1, R2, R3              // R1 = R2 & R3

2:          AND R1, R2, 0x1234          // R1 = R2 & 0x1234

3:          .set value1, 0x03           // constant value1=0x03
            AND R1, R2, value1          // R1 = R2 & value1

4:          .global label               // declaration of variable label
            AND R1, R2, label           // R1 = R2 & label
              ...
  label:    nop                         // definition of variable label
```

OR - Bitwise logical OR of two operands

Syntax **OR** *Rdst, Rsrc1, Rsrc2*
OR *Rdst, Rsrc1, imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does a bitwise logical OR of a source register and another source register or a 16-bit signed value and stores the result to the destination register.

Examples:

```
1:          OR R1, R2, R3               // R1 = R2 || R3

2:          OR R1, R2, 0x1234           // R1 = R2 || 0x1234

3:          .set value1, 0x03           // constant value1=0x03
            OR R1, R2, value1           // R1 = R2 || value1
```

(continues on next page)

(continued from previous page)

```
4:        .global label           // declaration of variable label
          OR R1, R2, label        // R1 = R2 || label
          ...
  label: nop                      // definition of variable label
```

LSH - Logical Shift Left

Syntax **LSH** *Rdst, Rsrc1, Rsrc2*
LSH *Rdst, Rsrc1, imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does a logical shift to left of the source register by the number of bits from another source register or a 16-bit signed value and stores the result to the destination register.

Note: Shift operations which are greater than 15 bits will have an undefined result.

Examples:

```
1:        LSH R1, R2, R3            // R1 = R2 << R3

2:        LSH R1, R2, 0x03          // R1 = R2 << 0x03

3:        .set value1, 0x03         // constant value1=0x03
          LSH R1, R2, value1        // R1 = R2 << value1

4:        .global label             // declaration of variable label
          LSH R1, R2, label         // R1 = R2 << label
          ...
  label:  nop                       // definition of variable label
```

RSH - Logical Shift Right

Syntax **RSH** *Rdst, Rsrc1, Rsrc2*
RSH *Rdst, Rsrc1, imm*

Operands *Rdst* - Register R[0..3] *Rsrc1* - Register R[0..3] *Rsrc2* - Register R[0..3] *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does a logical shift to right of a source register by the number of bits from another source register or a 16-bit signed value and stores the result to the destination register.

Note: Shift operations which are greater than 15 bits will have an undefined result.

Examples:

```
1:        RSH R1, R2, R3            // R1 = R2 >> R3

2:        RSH R1, R2, 0x03          // R1 = R2 >> 0x03

3:        .set value1, 0x03         // constant value1=0x03
          RSH R1, R2, value1        // R1 = R2 >> value1
```

(continues on next page)

(continued from previous page)

```
4:        .global label              // declaration of variable label
          RSH R1, R2, label          // R1 = R2 >> label
  label:  nop                        // definition of variable label
```

MOVE –Move to register

Syntax **MOVE** *Rdst, Rsrc*
MOVE *Rdst, imm*

Operands

- *Rdst* –Register R[0..3]
- *Rsrc* –Register R[0..3]
- *Imm* –16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction moves the value from the source register or a 16-bit signed value to the destination register.

Note: Note that when a label is used as an immediate, the address of the label will be converted from bytes to words. This is because LD, ST, and JUMP instructions expect the address register value to be expressed in words rather than bytes. See the section *Note about addressing* for more details.

Examples:

```
1:        MOVE       R1, R2            // R1 = R2

2:        MOVE       R1, 0x03          // R1 = 0x03

3:        .set       value1, 0x03      // constant value1=0x03
          MOVE       R1, value1        // R1 = value1

4:        .global     label            // declaration of label
          MOVE        R1, label        // R1 = address_of(label) / 4
          ...
  label:  nop                          // definition of label
```

ST –Store data to the memory

Syntax **ST** *Rsrc, Rdst, offset*

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Offset* –13-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores the 16-bit value of Rsrc to the lower half-word of memory with address Rdst+offset. The upper half-word is written with the current program counter (PC) (expressed in words, shifted left by 5 bits) OR' d with Rdst (0..3):

```
Mem[Rdst + offset / 4]{31:0} = {PC[10:0], 3'b0, Rdst, Rsrc[15:0]}
```

The application can use the higher 16 bits to determine which instruction in the ULP program has written any particular word into memory.

Note: Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section *Note about addressing* for more details.

Examples:

```
1:          ST   R1, R2, 0x12          // MEM[R2 + 0x12 / 4] = R1

2:          .data                      // Data section definition
  Addr1:    .word      123             // Define label Addr1 16 bit
            .set       offs, 0x00      // Define constant offs
            .text                      // Text section definition
            MOVE       R1, 1           // R1 = 1
            MOVE       R2, Addr1       // R2 = Addr1
            ST         R1, R2, offs    // MEM[R2 +  0 / 4] = R1
                                       // MEM[Addr1 + 0] will be 32'h600001
```

LD –Load data from the memory

Syntax **LD** *Rdst, Rsrc, offset*

Operands

- *Rdst* –Register R[0..3], destination
- *Rsrc* –Register R[0..3], holds address of destination, in 32-bit words
- *Offset* –13-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction loads the lower 16-bit half-word from memory with address [Rsrc + offset / 4] into the destination register Rdst:

```
Rdst[15:0] = Mem[Rsrc + offset / 4][15:0]
```

Note: Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section *Note about addressing* for more details.

Examples:

```
1:          LD   R1, R2, 0x12          // R1 = MEM[R2 + 0x12 / 4]

2:          .data                      // Data section definition
  Addr1:    .word      123             // Define label Addr1 16 bit
            .set       offs, 0x00      // Define constant offs
            .text                      // Text section definition
            MOVE       R1, 1           // R1 = 1
            MOVE       R2, Addr1       // R2 = Addr1 / 4 (address of label is␣
↪converted into words)
            LD         R1, R2, offs    // R1 = MEM[R2 +  0]
                                       // R1 will be 123
```

JUMP –Jump to an absolute address

Syntax **JUMP** *Rdst*

JUMP *ImmAddr*

JUMP *Rdst, Condition*

JUMP *ImmAddr, Condition*

Operands

- *Rdst* –Register R[0..3] containing address to jump to (expressed in 32-bit words)
- *ImmAddr* –13 bits address (expressed in bytes), aligned to 4 bytes
- ***Condition*:**
  - EQ –jump if last ALU operation result was zero
  - OV –jump if last ALU has set overflow flag

Cycles 2 cycles to execute, 2 cycles to fetch next instruction

Description The instruction makes jump to the specified address. Jump can be either unconditional or based on an ALU flag.

Examples:

```
1:          JUMP        R1              // Jump to address in R1 (address in R1 is in␣
↪32-bit words)

2:          JUMP        0x120, EQ       // Jump to address 0x120 (in bytes) if ALU␣
↪result is zero

3:          JUMP        label           // Jump to label
            ...
  label:    nop                         // Definition of label

4:          .global     label           // Declaration of global label

            MOVE        R1, label       // R1 = label (value loaded into R1 is in words)
            JUMP        R1              // Jump to label
            ...
  label:    nop                         // Definition of label
```

JUMPR –Jump to a relative offset (condition based on R0)

Syntax **JUMPR** *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- ***Condition*:**
  - *EQ* (equal) –jump if value in R0 == threshold
  - *LT* (less than) –jump if value in R0 < threshold
  - *LE* (less or equal) –jump if value in R0 <= threshold
  - *GT* (greater than) –jump if value in R0 > threshold
  - *GE* (greater or equal) –jump if value in R0 >= threshold

Cycles

Conditions *LT*, *GE*, *LE* and *GT*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *LE* and *GT* are implemented in the assembler using one **JUMPR** instruction:

```
// JUMPR target, threshold, GT is implemented as:

          JUMPR target, threshold+1, GE

// JUMPR target, threshold, LE is implemented as:

          JUMPR target, threshold + 1, LT
```

Conditions *EQ* is implemented in the assembler using two **JUMPR** instructions:

```
// JUMPR target, threshold, EQ is implemented as:

          JUMPR next, threshold + 1, GE
          JUMPR target, threshold, GE
next:
```

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of R0 register value and the threshold value.

Examples:

```
1:pos:    JUMPR       16, 20, GE    // Jump to address (position + 16 bytes) if␣
↪value in R0 >= 20
```

(continues on next page)

(continued from previous page)

```
2:        // Down counting loop using R0 register
          MOVE        R0, 16       // load 16 into R0
  label:  SUB         R0, R0, 1    // R0--
          NOP                      // do something
          JUMPR       label, 1, GE // jump to label if R0 >= 1
```

JUMPS –Jump to a relative address (condition based on stage count)

Syntax **JUMPS** *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- ***Condition*:**
  - *EQ* (equal) –jump if value in stage_cnt == threshold
  - *LT* (less than) –jump if value in stage_cnt < threshold
  - *LE* (less or equal) - jump if value in stage_cnt <= threshold
  - *GT* (greater than) –jump if value in stage_cnt > threshold
  - *GE* (greater or equal) —jump if value in stage_cnt >= threshold

Cycles

Conditions *LE*, *LT*, *GE*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *EQ*, *GT* are implemented in the assembler using two **JUMPS** instructions:

```
// JUMPS target, threshold, EQ is implemented as:

          JUMPS next, threshold, LT
          JUMPS target, threshold, LE
next:

// JUMPS target, threshold, GT is implemented as:

          JUMPS next, threshold, LE
          JUMPS target, threshold, GE
next:
```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of count register value and threshold value.

Examples:

```
1:pos:    JUMPS     16, 20, EQ     // Jump to (position + 16 bytes) if stage_cnt␣
↪== 20

2:        // Up counting loop using stage count register
          STAGE_RST                  // set stage_cnt to 0
  label:  STAGE_INC  1               // stage_cnt++
          NOP                        // do something
          JUMPS       label, 16, LT  // jump to label if stage_cnt < 16
```

STAGE_RST –Reset stage count register

Syntax **STAGE_RST**

Operands No operands

Description The instruction sets the stage count register to 0

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Examples:

```
1:        STAGE_RST       // Reset stage count register
```

STAGE_INC –Increment stage count register

Syntax **STAGE_INC** *Value*
Operands

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction
Description The instruction increments the stage count register by the given value.

Examples:

```
1:        STAGE_INC       10          // stage_cnt += 10

2:        // Up counting loop example:
          STAGE_RST                   // set stage_cnt to 0
  label:  STAGE_INC  1                // stage_cnt++
          NOP                         // do something
          JUMPS       label, 16, LT   // jump to label if stage_cnt < 16
```

STAGE_DEC –Decrement stage count register

Syntax **STAGE_DEC** *Value*
Operands

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction
Description The instruction decrements the stage count register by the given value.

Examples:

```
1:        STAGE_DEC       10        // stage_cnt -= 10;

2:        // Down counting loop example
          STAGE_RST                 // set stage_cnt to 0
          STAGE_INC  16             // increment stage_cnt to 16
  label:  STAGE_DEC  1              // stage_cnt--;
          NOP                       // do something
          JUMPS       label, 0, GT  // jump to label if stage_cnt > 0
```

HALT –End the program

Syntax **HALT**
Operands No operands
Cycles 2 cycles to execute
Description The instruction halts the ULP coprocessor and restarts the ULP wakeup timer, if it is enabled.

Examples:

```
1:        HALT       // Halt the coprocessor
```

WAKE –Wake up the chip

Syntax **WAKE**
Operands No operands
Cycles 2 cycles to execute, 4 cycles to fetch next instruction
Description The instruction sends an interrupt from the ULP coprocessor to the RTC controller.

- If the SoC is in deep sleep mode, and ULP wakeup is enabled, this causes the SoC to wake up.

- If the SoC is not in deep sleep mode, and ULP interrupt bit (RTC_CNTL_ULP_CP_INT_ENA) is set in RTC_CNTL_INT_ENA_REG register, RTC interrupt will be triggered.

Note that before using WAKE instruction, ULP program may needs to wait until RTC controller is ready to wake up the main CPU. This is indicated using RTC_CNTL_RDY_FOR_WAKEUP bit of RTC_CNTL_LOW_POWER_ST_REG register. If WAKE instruction is executed while RTC_CNTL_RDY_FOR_WAKEUP is zero, it has no effect (wake up does not occur).

Examples:

```
1: is_rdy_for_wakeup:                    // Read RTC_CNTL_RDY_FOR_WAKEUP bit
          READ_RTC_FIELD(RTC_CNTL_LOW_POWER_ST_REG, RTC_CNTL_RDY_FOR_WAKEUP)
          AND r0, r0, 1
          JUMP is_rdy_for_wakeup, eq     // Retry until the bit is set
          WAKE                           // Trigger wake up
          REG_WR 0x006, 24, 24, 0        // Stop ULP timer (clear RTC_CNTL_ULP_CP_
↪SLP_TIMER_EN)
          HALT                           // Stop the ULP program
          // After these instructions, SoC will wake up,
          // and ULP will not run again until started by the main program.
```

SLEEP –set ULP wakeup timer period

Syntax **SLEEP** *sleep_reg*

Operands

- *sleep_reg* –0..4, selects one of SENS_ULP_CP_SLEEP_CYCx_REG registers.

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction selects which of the SENS_ULP_CP_SLEEP_CYCx_REG (x = 0..4) register values is to be used by the ULP wakeup timer as wakeup period. By default, the value from SENS_ULP_CP_SLEEP_CYC0_REG is used.

Examples:

```
1:        SLEEP     1          // Use period set in SENS_ULP_CP_SLEEP_CYC1_REG

2:        .set sleep_reg, 4    // Set constant
          SLEEP  sleep_reg     // Use period set in SENS_ULP_CP_SLEEP_CYC4_REG
```

WAIT –wait some number of cycles

Syntax **WAIT** *Cycles*

Operands

- *Cycles* –number of cycles for wait

Cycles 2 + *Cycles* cycles to execute, 4 cycles to fetch next instruction

Description The instruction delays for given number of cycles.

Examples:

```
1:        WAIT     10          // Do nothing for 10 cycles

2:        .set  wait_cnt, 10   // Set a constant
          WAIT  wait_cnt       // wait for 10 cycles
```

ADC –do measurement with ADC

Syntax

- **ADC** *Rdst, Sar_sel, Mux*
- **ADC** *Rdst, Sar_sel, Mux, 0* —deprecated form

Operands

- *Rdst* –Destination Register R[0..3], result will be stored to this register
- *Sar_sel* –Select ADC: 0 = SARADC1, 1 = SARADC2

- *Mux* - Enable ADC channel. Channel number is [Mux-1]. If the user passes Mux value 1, then ADC channel 0 gets used.

Cycles `23 + max(1, SAR_AMP_WAIT1) + max(1, SAR_AMP_WAIT2) + max(1, SAR_AMP_WAIT3) + SARx_SAMPLE_CYCLE + SARx_SAMPLE_BIT` cycles to execute, 4 cycles to fetch next instruction

Description The instruction makes measurements from ADC.

Examples:

```
.. only:: esp32
```

1: ADC R1, 0, 1 // Measure value using ADC1 channel 0 and store result into R1

I2C_RD - read single byte from I2C slave

Syntax

- **I2C_RD** *Sub_addr, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to read.
- *High*, *Low* —Define range of bits to read. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description `I2C_RD` instruction reads one byte from I2C slave with index `Slave_sel`. Slave address (in 7-bit format) has to be set in advance into *SENS_I2C_SLAVE_ADDRx* register field, where `x == Slave_sel`. 8 bits of read result is stored into *R0* register.

Examples:

```
1:        I2C_RD      0x10, 7, 0, 0      // Read byte from sub-address 0x10 of␣
↪slave with address set in SENS_I2C_SLAVE_ADDR0
```

I2C_WR - write single byte to I2C slave

Syntax

- **I2C_WR** *Sub_addr, Value, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to write.
- *Value* –8-bit value to be written.
- *High*, *Low* —Define range of bits to write. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description `I2C_WR` instruction writes one byte to I2C slave with index `Slave_sel`. Slave address (in 7-bit format) has to be set in advance into *SENS_I2C_SLAVE_ADDRx* register field, where `x == Slave_sel`.

Examples:

```
1:        I2C_WR      0x20, 0x33, 7, 0, 1      // Write byte 0x33 to sub-address␣
↪0x20 of slave with address set in SENS_I2C_SLAVE_ADDR1.
```

REG_RD –read from peripheral register

Syntax **REG_RD** *Addr, High, Low*

Operands

- *Addr* –Register address, in 32-bit words
- *High* –Register end bit number
- *Low* –Register start bit number

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction reads up to 16 bits from a peripheral register into a general purpose register: `R0 = REG[Addr][High:Low]`.

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_RTCCNTL_BASE) / 4
```

Examples:

```
1:        REG_RD      0x120, 7, 4     // load 4 bits: R0 = {12'b0, REG[0x120][7:4]}
```

REG_WR –write to peripheral register

Syntax **REG_WR** *Addr, High, Low, Data*

Operands

- *Addr* –Register address, in 32-bit words.
- *High* –Register end bit number
- *Low* –Register start bit number
- *Data* –Value to write, 8 bits

Cycles 8 cycles to execute, 4 cycles to fetch next instruction

Description The instruction writes up to 8 bits from an immediate data value into a peripheral register: `REG[Addr][High:Low] = data`.

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_RTCCNTL_BASE) / 4
```

Examples:

```
1:        REG_WR      0x120, 7, 0, 0x10   // set 8 bits: REG[0x120][7:0] = 0x10
```

Convenience macros for peripheral registers access ULP source files are passed through C preprocessor before the assembler. This allows certain macros to be used to facilitate access to peripheral registers.

Some existing macros are defined in `soc/soc_ulp.h` header file. These macros allow access to the fields of peripheral registers by their names. Peripheral registers names which can be used with these macros are the ones defined in `soc/rtc_cntl_reg.h`, `soc/rtc_io_reg.h`, `soc/sens_reg.h`, and `soc/rtc_i2c_reg.h`.

READ_RTC_REG(rtc_reg, low_bit, bit_width) Read up to 16 bits from rtc_reg[low_bit + bit_width - 1 : low_bit] into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Read 16 lower bits of RTC_CNTL_TIME0_REG into R0 */
READ_RTC_REG(RTC_CNTL_TIME0_REG, 0, 16)
```

READ_RTC_FIELD(rtc_reg, field) Read from a field in rtc_reg into R0, up to 16 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/sens_reg.h"

/* Read 8-bit SENS_TSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSENS_OUT)
```

WRITE_RTC_REG(rtc_reg, low_bit, bit_width, value) Write immediate value into rtc_reg[low_bit + bit_width - 1 : low_bit], bit_width <= 8. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"

/* Set BIT(2) of RTC_GPIO_OUT_DATA_W1TS field in RTC_GPIO_OUT_W1TS_REG */
WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S + 2, 1, 1)
```

WRITE_RTC_FIELD(rtc_reg, field, value) Write immediate value into a field in rtc_reg, up to 8 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

Programming ULP FSM coprocessor using C macros (legacy) In addition to the existing binutils port for the ESP32 ULP coprocessor, it is possible to generate programs for the ULP FSM coprocessor by embedding assembly-like macros into an ESP32 application. Here is an example how this can be done:

```
const ulp_insn_t program[] = {
    I_MOVI(R3, 16),          // R3 <- 16
    I_LD(R0, R3, 0),         // R0 <- RTC_SLOW_MEM[R3 + 0]
    I_LD(R1, R3, 1),         // R1 <- RTC_SLOW_MEM[R3 + 1]
    I_ADDR(R2, R0, R1),      // R2 <- R0 + R1
    I_ST(R2, R3, 2),         // R2 -> RTC_SLOW_MEM[R2 + 2]
    I_HALT()
};
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

The `program` array is an array of `ulp_insn_t`, i.e. ULP coprocessor instructions. Each `I_XXX` preprocessor define translates into a single 32-bit instruction. Arguments of these preprocessor defines can be register numbers (`R0`—`R3`) and literal constants. See the API reference section at the end of this guide for descriptions of instructions and arguments they take.

Note: Because some of the instruction macros expand to inline function calls, defining such array in global scope will cause the compiler to produce an "initializer element is not constant" error. To fix this error, move the definition of instructions array into local scope.

Note: Load, store and move instructions use **addresses expressed in 32-bit words**. Address 0 corresponds to the first word of `RTC_SLOW_MEM`. This is different to how address arguments are handled in assembly code of the same instructions. See the section *Note about addressing* for more details for reference.

To generate branch instructions, special `M_` preprocessor defines are used. `M_LABEL` define can be used to define a branch target. Label identifier is a 16-bit integer. `M_Bxxx` defines can be used to generate branch instructions with target set to a particular label.

Implementation note: these `M_` preprocessor defines will be translated into two ulp_insn_t values: one is a token value which contains label number, and the other is the actual instruction. `ulp_process_macros_and_load` function resolves the label number to the address, modifies the branch instruction to use the correct address, and removes the the extra `ulp_insn_t` token which contains the label numer.

Here is an example of using labels and branches:

```
const ulp_insn_t program[] = {
    I_MOVI(R0, 34),          // R0 <- 34
    M_LABEL(1),              // label_1
    I_MOVI(R1, 32),          // R1 <- 32
    I_LD(R1, R1, 0),         // R1 <- RTC_SLOW_MEM[R1]
    I_MOVI(R2, 33),          // R2 <- 33
    I_LD(R2, R2, 0),         // R2 <- RTC_SLOW_MEM[R2]
    I_SUBR(R3, R1, R2),      // R3 <- R1 - R2
    I_ST(R3, R0, 0),         // R3 -> RTC_SLOW_MEM[R0 + 0]
    I_ADDI(R0, R0, 1),       // R0++
    M_BL(1, 64),             // if (R0 < 64) goto label_1
    I_HALT(),
};
RTC_SLOW_MEM[32] = 42;
RTC_SLOW_MEM[33] = 18;
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

API Reference

Header File

- components/ulp/ulp_fsm/include/esp32/ulp.h

Functions

static inline uint32_t **SOC_REG_TO_ULP_PERIPH_SEL**(uint32_t reg)

Map SoC peripheral register to periph_sel field of RD_REG and WR_REG instructions.

Parameters **reg** –peripheral register in RTC_CNTL_, RTC_IO_, SENS_, RTC_I2C peripherals.

Returns periph_sel value for the peripheral to which this register belongs.

Unions

union **ulp_insn**

#include <ulp.h> Instruction format structure.

All ULP instructions are 32 bit long. This union contains field layouts used by all of the supported instructions. This union also includes a special "macro"instruction layout. This is not a real instruction which can be executed by the CPU. It acts as a token which is removed from the program by the ulp_process_macros_and_load function.

These structures are not intended to be used directly. Preprocessor definitions provided below fill the fields of these structure with the right arguments.

Public Members

uint32_t **cycles**

Number of cycles to sleep

TBD, cycles used for measurement

uint32_t **unused**

Unused

uint32_t **opcode**

Opcode (OPCODE_DELAY)

Opcode (OPCODE_ST)

Opcode (OPCODE_LD)

Opcode (OPCODE_HALT)

Opcode (OPCODE_BRANCH)

Opcode (OPCODE_ALU)

Opcode (OPCODE_WR_REG)

Opcode (OPCODE_RD_REG)

Opcode (OPCODE_ADC)

Opcode (OPCODE_TSENS)

Opcode (OPCODE_I2C)

Opcode (OPCODE_END)

Opcode (OPCODE_MACRO)

struct *ulp_insn*::[anonymous] **delay**

Format of DELAY instruction

uint32_t **dreg**

Register which contains data to store

Register where the data should be loaded to

Register which contains target PC, expressed in words (used if .reg == 1)

Destination register

Register where to store ADC result

Register where to store temperature measurement result

Destination register (for SUB_OPCODE_MACRO_LABELPC) >

uint32_t **sreg**

Register which contains address in RTC memory (expressed in words)

Register with operand A

uint32_t **unused1**

Unused

uint32_t **offset**

Offset to add to sreg

Absolute value of target PC offset w.r.t. current PC, expressed in words

uint32_t **unused2**

Unused

uint32_t **sub_opcode**

Sub opcode (SUB_OPCODE_ST)

Sub opcode (SUB_OPCODE_BX)

Sub opcode (SUB_OPCODE_B)

Sub opcode (SUB_OPCODE_BS)

Sub opcode (SUB_OPCODE_ALU_REG)

Sub opcode (SUB_OPCODE_ALU_CNT)

Sub opcode (SUB_OPCODE_ALU_IMM)

Sub opcode (SUB_OPCODE_WAKEUP)

Sub opcode (SUB_OPCODE_SLEEP)

SUB_OPCODE_MACRO_LABEL or SUB_OPCODE_MACRO_BRANCH or SUB_OPCODE_MACRO_LABELPC

struct *ulp_insn*::[anonymous] **st**

Format of ST instruction

struct *ulp_insn*::[anonymous] **ld**

Format of LD instruction

struct *ulp_insn*::[anonymous] **halt**

Format of HALT instruction

uint32_t **addr**

Target PC, expressed in words (used if .reg == 0)

Address within either RTC_CNTL, RTC_IO, or SARADC

uint32_t **reg**

Target PC in register (1) or immediate (0)

uint32_t **type**

Jump condition (BX_JUMP_TYPE_xxx)

struct *ulp_insn*::[anonymous] **bx**

Format of BRANCH instruction (absolute address)

uint32_t **imm**

Immediate value to compare against

Immediate value of operand

Immediate value of operand B

uint32_t **cmp**

Comparison to perform: B_CMP_L or B_CMP_GE

Comparison to perform: JUMPS_LT, JUMPS_GE or JUMPS_LE

uint32_t **sign**

Sign of target PC offset: 0: positive, 1: negative

struct *ulp_insn*::[anonymous] **b**

Format of BRANCH instruction (relative address, conditional on R0)

struct *ulp_insn*::[anonymous] **bs**

Format of BRANCH instruction (relative address, conditional on the stage counter)

uint32_t **treg**

Register with operand B

uint32_t **sel**

Operation to perform, one of ALU_SEL_xxx

Operation to perform, one of ALU_SEL_Sxxx

struct *ulp_insn*::[anonymous] **alu_reg**

Format of ALU instruction (both sources are registers)

struct *ulp_insn*::[anonymous] **alu_reg_s**

Format of ALU instruction (stage counter and an immediate)

struct *ulp_insn*::[anonymous] **alu_imm**

Format of ALU instruction (one source is an immediate)

uint32_t **periph_sel**

Select peripheral: RTC_CNTL (0), RTC_IO(1), SARADC(2)

uint32_t **data**

8 bits of data to write

8 bits of data for write operation

uint32_t **low**

Low bit

uint32_t **high**

High bit

struct *ulp_insn*::[anonymous] **wr_reg**

Format of WR_REG instruction

struct *ulp_insn*::[anonymous] **rd_reg**

Format of RD_REG instruction

uint32_t **mux**

Select SARADC pad (mux + 1)

uint32_t **sar_sel**

Select SARADC0 (0) or SARADC1 (1)

struct *ulp_insn*::[anonymous] **adc**

Format of ADC instruction

uint32_t **wait_delay**

Cycles to wait after measurement is done

uint32_t **reserved**

Reserved, set to 0

struct *ulp_insn*::[anonymous] **tsens**

Format of TSENS instruction

uint32_t **i2c_addr**

I2C slave address

uint32_t **low_bits**

low bit of range for write operation (lower bits are masked)

uint32_t **high_bits**

high bit of range for write operation (higher bits are masked)

uint32_t **i2c_sel**

index of slave address register [7:0]

uint32_t **rw**

Write (1) or read (0)

struct *ulp_insn*::[anonymous] **i2c**

Format of I2C instruction

uint32_t **wakeup**

Set to 1 to wake up chip

struct *ulp_insn*::[anonymous] **end**

Format of END instruction with wakeup

uint32_t **cycle_sel**

Select which one of SARADC_ULP_CP_SLEEP_CYCx_REG to get the sleep duration from

struct *ulp_insn*::[anonymous] **sleep**

Format of END instruction with sleep

uint32_t **label**

Label number

struct *ulp_insn*::[anonymous] **macro**

Format of tokens used by MACROs

uint32_t **instruction**

Encoded instruction for ULP coprocessor

Macros

R0

general purpose register 0

R1

general purpose register 1

R2

general purpose register 2

R3

general purpose register 3

OPCODE_WR_REG

Instruction: write peripheral register (RTC_CNTL/RTC_IO/SARADC)

OPCODE_RD_REG

Instruction: read peripheral register (RTC_CNTL/RTC_IO/SARADC)

RD_REG_PERIPH_RTC_CNTL

Identifier of RTC_CNTL peripheral for RD_REG and WR_REG instructions

RD_REG_PERIPH_RTC_IO

Identifier of RTC_IO peripheral for RD_REG and WR_REG instructions

RD_REG_PERIPH_SENS

Identifier of SARADC peripheral for RD_REG and WR_REG instructions

RD_REG_PERIPH_RTC_I2C

Identifier of RTC_I2C peripheral for RD_REG and WR_REG instructions

OPCODE_I2C

Instruction: read/write I2C

SUB_OPCODE_I2C_RD

I2C read

SUB_OPCODE_I2C_WR

I2C write

OPCODE_DELAY

Instruction: delay (nop) for a given number of cycles

OPCODE_ADC

Instruction: SAR ADC measurement

OPCODE_ST

Instruction: store indirect to RTC memory

SUB_OPCODE_ST

Store 32 bits, 16 MSBs contain PC, 16 LSBs contain value from source register

OPCODE_ALU

Arithmetic instructions

SUB_OPCODE_ALU_REG

Arithmetic instruction, both source values are in register

SUB_OPCODE_ALU_IMM

Arithmetic instruction, one source value is an immediate

SUB_OPCODE_ALU_CNT

Arithmetic instruction, stage counter and an immediate

ALU_SEL_ADD

Addition

ALU_SEL_SUB

Subtraction

ALU_SEL_AND

Logical AND

ALU_SEL_OR

Logical OR

ALU_SEL_MOV

Copy value (immediate to destination register or source register to destination register

ALU_SEL_LSH

Shift left by given number of bits

ALU_SEL_RSH

Shift right by given number of bits

ALU_SEL_SINC

Increment the stage counter

ALU_SEL_SDEC

Decrement the stage counter

ALU_SEL_SRST

Reset the stage counter

OPCODE_BRANCH

Branch instructions

SUB_OPCODE_BX

Branch to absolute PC (immediate or in register)

SUB_OPCODE_BR

Branch to relative PC, conditional on R0

SUB_OPCODE_BS

Branch to relative PC, conditional on the stage counter

BX_JUMP_TYPE_DIRECT

Unconditional jump

BX_JUMP_TYPE_ZERO

Branch if last ALU result is zero

BX_JUMP_TYPE_OVF

Branch if last ALU operation caused and overflow

SUB_OPCODE_B

Branch to a relative offset

B_CMP_L

Branch if R0 is less than an immediate

B_CMP_GE

Branch if R0 is greater than or equal to an immediate

JUMPS_LT

Branch if the stage counter <

JUMPS_GE

Branch if the stage counter >=

JUMPS_LE

Branch if the stage counter <=

OPCODE_END

Stop executing the program

SUB_OPCODE_END

Stop executing the program and optionally wake up the chip

SUB_OPCODE_SLEEP

Stop executing the program and run it again after selected interval

OPCODE_TSENS

Instruction: temperature sensor measurement. Poor accuracy, not recommended for most use-cases

OPCODE_HALT

Halt the coprocessor

OPCODE_LD

Indirect load lower 16 bits from RTC memory

OPCODE_MACRO

Not a real opcode. Used to identify labels and branches in the program

SUB_OPCODE_MACRO_LABEL

Label macro

SUB_OPCODE_MACRO_BRANCH

Branch macro

SUB_OPCODE_MACRO_LABELPC

Label pointer macro

I_DELAY(cycles_)

Delay (nop) for a given number of cycles

I_HALT()

Halt the coprocessor.

This instruction halts the coprocessor, but keeps ULP timer active. As such, ULP program will be restarted again by timer. To stop the program and prevent the timer from restarting the program, use I_END(0) instruction.

I_WR_REG(reg, low_bit, high_bit, val)

Write literal value to a peripheral register

reg[high_bit : low_bit] = val This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_RD_REG(reg, low_bit, high_bit)

Read from peripheral register into R0

R0 = reg[high_bit : low_bit] This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_WR_REG_BIT(reg, shift, val)

Set or clear a bit in the peripheral register.

Sets bit (1 << shift) of register reg to value val. This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_WAKE()

Wake the SoC from deep sleep.

This instruction initiates wake up from deep sleep. Use esp_deep_sleep_enable_ulp_wakeup to enable deep sleep wakeup triggered by the ULP before going into deep sleep. Note that ULP program will still keep running

until the I_HALT instruction, and it will still be restarted by timer at regular intervals, even when the SoC is woken up.

To stop the ULP program, use I_HALT instruction.

To disable the timer which start ULP program, use I_END() instruction. I_END instruction clears the RTC_CNTL_ULP_CP_SLP_TIMER_EN_S bit of RTC_CNTL_STATE0_REG register, which controls the ULP timer.

I_END()

Stop ULP program timer.

This is a convenience macro which disables the ULP program timer. Once this instruction is used, ULP program will not be restarted anymore until ulp_run function is called.

ULP program will continue running after this instruction. To stop the currently running program, use I_HALT().

I_SLEEP_CYCLE_SEL(timer_idx)

Select the time interval used to run ULP program.

This instructions selects which of the SENS_SLEEP_CYCLES_Sx registers' value is used by the ULP program timer. When the ULP program stops at I_HALT instruction, ULP program timer start counting. When the counter reaches the value of the selected SENS_SLEEP_CYCLES_Sx register, ULP program start running again from the start address (passed to the ulp_run function). There are 5 SENS_SLEEP_CYCLES_Sx registers, so 0 <= timer_idx < 5.

By default, SENS_SLEEP_CYCLES_S0 register is used by the ULP program timer.

I_TSENS(reg_dest, delay)

Perform temperature sensor measurement and store it into reg_dest.

Delay can be set between 1 and ((1 << 14) - 1). Higher values give higher measurement resolution.

I_ADC(reg_dest, adc_idx, pad_idx)

Perform ADC measurement and store result in reg_dest.

adc_idx selects ADC (0 or 1). pad_idx selects ADC pad (0 - 7).

I_ST(reg_val, reg_addr, offset_)

Store value from register reg_val into RTC memory.

The value is written to an offset calculated by adding value of reg_addr register and offset_ field (this offset is expressed in 32-bit words). 32 bits written to RTC memory are built as follows:

- bits [31:21] hold the PC of current instruction, expressed in 32-bit words
- bits [20:18] = 3'b0
- bits [17:16] reg_addr (0..3)
- bits [15:0] are assigned the contents of reg_val

RTC_SLOW_MEM[addr + offset_] = { insn_PC[10:0], 3'b0, reg_addr, reg_val[15:0] }

I_LD(reg_dest, reg_addr, offset_)

Load value from RTC memory into reg_dest register.

Loads 16 LSBs from RTC memory word given by the sum of value in reg_addr and value of offset_.

I_BL(pc_offset, imm_value)

Branch relative if R0 less than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BGE(pc_offset, imm_value)

Branch relative if R0 greater or equal than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BXR (reg_pc)

Unconditional branch to absolute PC, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXI (imm_pc)

Unconditional branch to absolute PC, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXZR (reg_pc)

Branch to absolute PC if ALU result is zero, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXZI (imm_pc)

Branch to absolute PC if ALU result is zero, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXFR (reg_pc)

Branch to absolute PC if ALU overflow, address in register

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXFI (imm_pc)

Branch to absolute PC if ALU overflow, immediate address

Address imm_pc is expressed in 32-bit words.

I_ADDR (reg_dest, reg_src1, reg_src2)

Addition: dest = src1 + src2

I_SUBR (reg_dest, reg_src1, reg_src2)

Subtraction: dest = src1 - src2

I_ANDR (reg_dest, reg_src1, reg_src2)

Logical AND: dest = src1 & src2

I_ORR (reg_dest, reg_src1, reg_src2)

Logical OR: dest = src1 | src2

I_MOVR (reg_dest, reg_src)

Copy: dest = src

I_LSHR (reg_dest, reg_src, reg_shift)

Logical shift left: dest = src << shift

I_RSHR (reg_dest, reg_src, reg_shift)

Logical shift right: dest = src >> shift

I_ADDI (reg_dest, reg_src, imm_)

Add register and an immediate value: dest = src1 + imm

I_SUBI (reg_dest, reg_src, imm_)

Subtract register and an immediate value: dest = src - imm

I_ANDI (reg_dest, reg_src, imm_)

Logical AND register and an immediate value: dest = src & imm

I_ORI (reg_dest, reg_src, imm_)

Logical OR register and an immediate value: dest = src | imm

I_MOVI (reg_dest, imm_)

Copy an immediate value into register: dest = imm

I_LSHI (reg_dest, reg_src, imm_)

Logical shift left register value by an immediate: dest = src << imm

I_RSHI (reg_dest, reg_src, imm_)

Logical shift right register value by an immediate: dest = val >> imm

M_LABEL (label_num)

Define a label with number label_num.

This is a macro which doesn't generate a real instruction. The token generated by this macro is removed by ulp_process_macros_and_load function. Label defined using this macro can be used in branch macros defined below.

M_BRANCH (label_num)

Token macro used by M_B and M_BX macros. Not to be used directly.

M_LABELPC (label_num)

Token macro used by M_MOVL macro. Not to be used directly.

M_MOVL (reg_dest, label_num)

Macro: Move the program counter at the given label into the register. This address can then be used with I_BXR, I_BXZR, I_BXFR, etc.

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BL (label_num, imm_value)

Macro: branch to label label_num if R0 is less than immediate value.

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BGE (label_num, imm_value)

Macro: branch to label label_num if R0 is greater or equal than immediate value

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BX (label_num)

Macro: unconditional branch to label

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BXZ (label_num)

Macro: branch to label if ALU result is zero

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BXF (label_num)

Macro: branch to label if ALU overflow

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

I_STAGE_INC (imm_)

Increment the stage counter by immediate value

I_STAGE_DEC(imm_)

Decrement the stage counter by immediate value

I_STAGE_RST()

Reset the stage counter

M_BSLT(label_num, imm_value)

Macro: branch to label if the stage counter is less than immediate value

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BSGE(label_num, imm_value)

Macro: branch to label if the stage counter is greater than or equal to immediate value

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BSLE(label_num, imm_value)

Macro: branch to label if the stage counter is less than or equal to immediate value

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BSEQ(label_num, imm_value)

Macro: branch to label if the stage counter is equal to immediate value. Implemented using two JUMPS instructions: JUMPS next, imm_value, LT JUMPS label_num, imm_value, LE

This macro generates three ulp_insn_t values separated by commas, and should be used when defining contents of ulp_insn_t arrays. Second value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BSGT(label_num, imm_value)

Macro: branch to label if the stage counter is greater than immediate value. Implemented using two instructions: JUMPS next, imm_value, LE JUMPS label_num, imm_value, GE

This macro generates three ulp_insn_t values separated by commas, and should be used when defining contents of ulp_insn_t arrays. Second value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

I_JUMPS(pc_offset, imm_value, comp_type)

Branch relative if (stage counter [comp_type] [imm_value]) evaluates to true.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is an 8-bit value to compare the stage counter against comp_type is the type of comparison to perform: JUMPS_LT (<), JUMPS_GE (>=) or JUMPS_LE (<=)

I_I2C_RW(sub_addr, val, low_bit, high_bit, slave_sel, rw_bit)

Perform an I2C transaction with a slave device. I_I2C_READ and I_I2C_WRITE are provided for convenience, instead of using this directly.

Slave address (in 7-bit format) has to be set in advance into SENS_I2C_SLAVE_ADDRx register field, where x == slave_sel. For read operations, 8 bits of read result is stored into R0 register. For write operations, val will be written to sub_addr at [high_bit:low_bit]. Bits outside of this range are masked.

I_I2C_READ(slave_sel, sub_addr)

Read a byte from the sub address of an I2C slave, and store the result in R0.

Slave address (in 7-bit format) has to be set in advance into SENS_I2C_SLAVE_ADDRx register field, where x == slave_sel.

I_I2C_WRITE (slave_sel, sub_addr, val)

Write a byte to the sub address of an I2C slave.

Slave address (in 7-bit format) has to be set in advance into SENS_I2C_SLAVE_ADDRx register field, where x == slave_sel.

Compiling the ULP Code

To compile the ULP FSM code as part of the component, the following steps must be taken:

1. The ULP FSM code, written in assembly, must be added to one or more files with *.S* extension. These files must be placed into a separate directory inside the component directory, for instance, *ulp/*.

Note: When registering the component (via `idf_component_register`), this directory should not be added to the `SRC_DIRS` argument. The logic behind this is that the ESP-IDF build system will compile files found in `SRC_DIRS` based on their extensions. For `.S` files, `xtensa-esp32-elf-as` assembler is used. This is not desirable for ULP FSM assembly files, so the easiest way to achieve the distinction is by placing ULP FSM assembly files into a separate directory. The ULP FSM assembly source files should also **not** be added to `SRCS` for the same reason. See the step below for how to properly add ULP FSM assembly source files.

---

2. Call `ulp_embed_binary` from the component CMakeLists.txt after registration. For example:

```
...
idf_component_register()

set(ulp_app_name ulp_${COMPONENT_NAME})
set(ulp_s_sources ulp/ulp_assembly_source_file.S)
set(ulp_exp_dep_srcs "ulp_c_source_file.c")

ulp_embed_binary(${ulp_app_name} "${ulp_s_sources}" "${ulp_exp_dep_srcs}")
```

The first argument to `ulp_embed_binary` specifies the ULP FSM binary name. The name specified here will also be used by other generated artifacts such as the ELF file, map file, header file and linker export file. The second argument specifies the ULP FSM assembly source files. Finally, the third argument specifies the list of component source files which include the header file to be generated. This list is needed to build the dependencies correctly and ensure that the generated header file will be created before any of these files are compiled. See the section below for the concept of generated header files for ULP applications.

3. Build the application as usual (e.g. *idf.py app*).
   Inside, the build system will take the following steps to build ULP FSM program:
   1. **Run each assembly file (foo.S) through the C preprocessor.** This step generates the preprocessed assembly files (foo.ulp.S) in the component build directory. This step also generates dependency files (foo.ulp.d).
   2. **Run preprocessed assembly sources through the assembler.** This produces object (foo.ulp.o) and listing (foo.ulp.lst) files. Listing files are generated for debugging purposes and are not used at later stages of the build process.
   3. **Run the linker script template through the C preprocessor.** The template is located in `components/ulp/ld` directory.
   4. **Link the object files into an output ELF file** (`ulp_app_name.elf`). The Map file (`ulp_app_name.map`) generated at this stage may be useful for debugging purposes.
   5. **Dump the contents of the ELF file into a binary** (`ulp_app_name.bin`) which can then be embedded into the application.
   6. **Generate a list of global symbols** (`ulp_app_name.sym`) in the ELF file using `esp32ulp-elf-nm`.
   7. **Create an LD export script and a header file** (`ulp_app_name.ld` and `ulp_app_name.h`) containing the symbols from `ulp_app_name.sym`. This is done using the `esp32ulp_mapgen.py` utility.
   8. **Add the generated binary to the list of binary files** to be embedded into the application.

Accessing the ULP FSM Program Variables

Global symbols defined in the ULP FSM program may be used inside the main program.

For example, the ULP FSM program may define a variable `measurement_count` which will define the number of ADC measurements the program needs to make before waking up the chip from deep sleep:

```
                        .global measurement_count
measurement_count:      .long 0

                        // later, use measurement_count
                        move r3, measurement_count
                        ld r3, r3, 0
```

The main program needs to initialize this variable before the ULP program is started. The build system makes this possible by generating the `${ULP_APP_NAME}.h` and `${ULP_APP_NAME}.ld` files which define the global symbols present in the ULP program. Each global symbol defined in the ULP program is included in these files and are prefixed with `ulp_`.

The header file contains the declaration of the symbol:

```
extern uint32_t ulp_measurement_count;
```

Note that all symbols (variables, arrays, functions) are declared as `uint32_t`. For functions and arrays, take the address of the symbol and cast it to the appropriate type.

The generated linker script file defines the locations of symbols in RTC_SLOW_MEM:

```
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access the ULP program variables from the main program, the generated header file should be included using an `include` statement. This will allow the ULP program variables to be accessed as regular variables:

```
#include "ulp_app_name.h"

// later
void init_ulp_vars() {
    ulp_measurement_count = 64;
}
```

Note that the ULP FSM program can only use the lower 16 bits of each 32-bit word in RTC memory, because the registers are 16-bit, and there is no instruction to load from the high part of the word. Likewise, the ULP store instruction writes register values into the lower 16 bits of the 32-bit word in RTC memory. The upper 16 bits are written with a value which depends on the address of the store instruction, thus when reading variables written by the ULP coprocessor, the main application needs to mask the upper 16 bits, e.g.:

```
printf("Last measurement value: %d\n", ulp_last_measurement & UINT16_MAX);
```

Starting the ULP FSM Program

To run a ULP FSM program, the main application needs to load the ULP program into RTC memory using the *`ulp_load_binary()`* function, and then start it using the *`ulp_run()`* function.

Note that the `Enable Ultra Low Power (ULP) Coprocessor` option must be enabled in menuconfig to work with ULP. To select the type of ULP to be used, the `ULP Co-processor type` option must be set. To reserve memory for the ULP, the `RTC slow memory reserved for coprocessor` option must be set to a value big enough to store ULP code and data. If the application components contain multiple ULP programs, then the size of the RTC memory must be sufficient to hold the largest one.

Each ULP program is embedded into the ESP-IDF application as a binary blob. The application can reference this blob and load it in the following way (suppose ULP_APP_NAME was defined to `ulp_app_name`):

```
extern const uint8_t bin_start[] asm("_binary_ulp_app_name_bin_start");
extern const uint8_t bin_end[]   asm("_binary_ulp_app_name_bin_end");

void start_ulp_program() {
    ESP_ERROR_CHECK( ulp_load_binary(
        0 // load address, set to 0 when using default linker scripts
        bin_start,
        (bin_end - bin_start) / sizeof(uint32_t)) );
}
```

Once the program is loaded into RTC memory, the application can start it by passing the address of the entry point to the `ulp_run` function:

```
ESP_ERROR_CHECK( ulp_run(&ulp_entry - RTC_SLOW_MEM) );
```

Declaration of the entry point symbol comes from the generated header file mentioned above, `${ULP_APP_NAME}.h`. In the assembly source of the ULP FSM application, this symbol must be marked as `.global`:

```
        .global entry
entry:
        // code starts here
```

ESP32 ULP program flow

ESP32 ULP coprocessor is started by a timer. The timer is started once *`ulp_run()`* is called. The timer counts a number of RTC_SLOW_CLK ticks (by default, produced by an internal 150 kHz RC oscillator). The number of ticks is set using `SENS_ULP_CP_SLEEP_CYCx_REG` registers (x = 0..4). When starting the ULP for the first time, `SENS_ULP_CP_SLEEP_CYC0_REG` will be used to set the number of timer ticks. Later the ULP program can select another `SENS_ULP_CP_SLEEP_CYCx_REG` register using `sleep` instruction.

The application can set ULP timer period values (SENS_ULP_CP_SLEEP_CYCx_REG, x = 0..4) using `ulp_set_wakeup_period` function.

Once the timer counts the number of ticks set in the selected `SENS_ULP_CP_SLEEP_CYCx_REG` register, ULP coprocessor powers up and starts running the program from the entry point set in the call to *`ulp_run()`*.

The program runs until it encounters a `halt` instruction or an illegal instruction. Once the program halts the ULP coprocessor powers down and the timer is started again.

To disable the timer (effectively preventing the ULP program from running again), clear the `RTC_CNTL_ULP_CP_SLP_TIMER_EN` bit in the `RTC_CNTL_STATE0_REG` register. This can be done both from ULP code and from the main program.

Application Examples

- ULP FSM Coprocessor counts pulses on an IO while main CPU is in deep sleep: system/ulp_fsm/ulp.
- ULP FSM Coprocessor polls ADC in while main CPU is in deep sleep: system/ulp_fsm/ulp_adc.

API Reference

Header File

- components/ulp/ulp_fsm/include/ulp_fsm_common.h

Functions

esp_err_t **ulp_process_macros_and_load**(uint32_t load_addr, const *ulp_insn_t* *program, size_t *psize)

Resolve all macro references in a program and load it into RTC memory.

Parameters
- **load_addr** –address where the program should be loaded, expressed in 32-bit words
- **program** –ulp_insn_t array with the program
- **psize** –size of the program, expressed in 32-bit words

Returns
- ESP_OK on success
- ESP_ERR_NO_MEM if auxiliary temporary structure can not be allocated
- one of ESP_ERR_ULP_xxx if program is not valid or can not be loaded

esp_err_t **ulp_load_binary**(uint32_t load_addr, const uint8_t *program_binary, size_t program_size)

Load ULP program binary into RTC memory.

ULP program binary should have the following format (all values little-endian):

a. MAGIC, (value 0x00706c75, 4 bytes)
b. TEXT_OFFSET, offset of .text section from binary start (2 bytes)
c. TEXT_SIZE, size of .text section (2 bytes)
d. DATA_SIZE, size of .data section (2 bytes)
e. BSS_SIZE, size of .bss section (2 bytes)
f. (TEXT_OFFSET - 12) bytes of arbitrary data (will not be loaded into RTC memory)
g. .text section
h. .data section

Linker script in components/ulp/ld/esp32.ulp.ld produces ELF files which correspond to this format. This linker script produces binaries with load_addr == 0.

Parameters
- **load_addr** –address where the program should be loaded, expressed in 32-bit words
- **program_binary** –pointer to program binary
- **program_size** –size of the program binary

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if load_addr is out of range
- ESP_ERR_INVALID_SIZE if program_size doesn't match (TEXT_OFFSET + TEXT_SIZE + DATA_SIZE)
- ESP_ERR_NOT_SUPPORTED if the magic number is incorrect

esp_err_t **ulp_run**(uint32_t entry_point)

Run the program loaded into RTC memory.

Parameters **entry_point** –entry point, expressed in 32-bit words
Returns ESP_OK on success

Macros

ESP_ERR_ULP_BASE

Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG

Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR

Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL

More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL

Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE

Branch target is out of range of B instruction (try replacing with BX)

Type Definitions

typedef union *ulp_insn* **ulp_insn_t**

Header File

- components/ulp/ulp_common/include/ulp_common.h

Functions

esp_err_t **ulp_set_wakeup_period**(size_t period_index, uint32_t period_us)

Set one of ULP wakeup period values.

ULP coprocessor starts running the program when the wakeup timer counts up to a given value (called period). There are 5 period values which can be programmed into SENS_ULP_CP_SLEEP_CYCx_REG registers, x = 0..4 for ESP32, and one period value which can be programmed into RTC_CNTL_ULP_CP_TIMER_1_REG register for ESP32-S2/S3. By default, for ESP32, wakeup timer will use the period set into SENS_ULP_CP_SLEEP_CYC0_REG, i.e. period number 0. ULP program code can use SLEEP instruction to select which of the SENS_ULP_CP_SLEEP_CYCx_REG should be used for subsequent wakeups.

However, please note that SLEEP instruction issued (from ULP program) while the system is in deep sleep mode does not have effect, and sleep cycle count 0 is used.

For ESP32-S2/S3 the SLEEP instruction not exist. Instead a WAKE instruction will be used.

Note: The ULP FSM requires two clock cycles to wakeup before being able to run the program. Then additional 16 cycles are reserved after wakeup waiting until the 8M clock is stable. The FSM also requires two more clock cycles to go to sleep after the program execution is halted. The minimum wakeup period that may be set up for the ULP is equal to the total number of cycles spent on the above internal tasks. For a default configuration of the ULP running at 150kHz it makes about 133us.

Parameters

- **period_index** –wakeup period setting number (0 - 4)
- **period_us** –wakeup period, us

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if period_index is out of range

void **ulp_timer_stop**(void)

Stop the ULP timer.

Note: This will stop the ULP from waking up if halted, but will not abort any program currently executing on the ULP.

---

void **ulp_timer_resume**(void)

Resume the ULP timer.

Note: This will resume an already configured timer, but does no other configuration

Header File

- components/ulp/ulp_common/include/esp32/ulp_common_defs.h

Macros

RTC_SLOW_MEM

RTC slow memory, 8k size

2.10.31 Watchdogs

Overview

The ESP-IDF has support for multiple types of watchdogs, with the two main ones being: The Interrupt Watchdog Timer and the Task Watchdog Timer (TWDT). The Interrupt Watchdog Timer and the TWDT can both be enabled using *Project Configuration Menu*, however the TWDT can also be enabled during runtime. The Interrupt Watchdog is responsible for detecting instances where FreeRTOS task switching is blocked for a prolonged period of time. The TWDT is responsible for detecting instances of tasks running without yielding for a prolonged period.

ESP-IDF has support for the following types of watchdog timers:

- Interrupt Watchdog Timer (IWDT)
- Task Watchdog Timer (TWDT)

The various watchdog timers can be enabled using the *Project Configuration Menu*. However, the TWDT can also be enabled during runtime.

Interrupt Watchdog Timer (IWDT)

The purpose of the IWDT is to ensure that interrupt service routines (ISRs) are not blocked from running for a prolonged period of time (i.e., the IWDT timeout period). Blocking ISRs from running in a timely manner is undesirable as it can increases ISR latency, and also prevents task switching (as task switching is executed form an ISR). The things that can block ISRs from running include:

- Disabling interrupts
- Critical Sections (also disables interrupts)
- Other same/higher priority ISRs (will block same/lower priority ISRs from running it completes execution)

The IWDT utilizes the watchdog timer in Timer Group 1 as its underlying hardware timer and leverages the FreeRTOS tick interrupt on each CPU to feed the watchdog timer. If the tick interrupt on a particular CPU is not run at within the IWDT timeout period, it is indicative that something is blocking ISRs from being run on that CPU (see the list of reasons above).

When the IWDT times out, the default action is to invoke the panic handler and display the panic reason as `Interrupt wdt timeout on CPU0` or `Interrupt wdt timeout on CPU1` (as applicable). Depending on the panic handler's configured behavior (see *CONFIG_ESP_SYSTEM_PANIC*), users can then debug the source of the IWDT timeout (via the backtrace, OpenOCD, gdbstub etc) or simply reset the chip (which may be preferred in a production environment).

If for whatever reason the panic handler is unable to run after an IWDT timeout, the IWDT has a secondary timeout that will hard-reset the chip (i.e., a system reset).

Configuration

- The IWDT is enabled by default via the *CONFIG_ESP_INT_WDT* option.
- The IWDT's timeout is configured by setting the *CONFIG_ESP_INT_WDT_TIMEOUT_MS* option.
  - Note that the default timeout is higher if PSRAM support is enabled, as a critical section or interrupt routine that accesses a large amount of PSRAM will take longer to complete in some circumstances.
  - The timeout should always at least twice longer than the period between FreeRTOS ticks (see *CONFIG_FREERTOS_HZ*).

Tuning If you find the IWDT timeout is triggered because an interrupt or critical section is running longer than the timeout period, consider rewriting the code:

- Critical sections should be made as short as possible. Any non-critical code/computation should be placed outside the critical section.
- Interrupt handlers should also perform the minimum possible amount of computation. Users can consider deferring any computation to a task by having the ISR push data to a task using queues.

Neither critical sections or interrupt handlers should ever block waiting for another event to occur. If changing the code to reduce the processing time is not possible or desirable, it's possible to increase the *CONFIG_ESP_INT_WDT_TIMEOUT_MS* setting instead.

Task Watchdog Timer (TWDT)

The Task Watchdog Timer (TWDT) is used to monitor particular tasks, ensuring that they are able to execute within a given timeout period. The TWDT primarily watches the Idle Tasks of each CPU, however any task can subscribe to be watched by the TWDT. By watching the Idle Tasks of each CPU, the TWDT can detect instances of tasks running for a prolonged period of time wihtout yielding. This can be an indicator of poorly written code that spinloops on a peripheral, or a task that is stuck in an infinite loop.

The TWDT is built around the Hardware Watchdog Timer in Timer Group 0. When a timeout occurs, an interrupt is triggered. Users can define the function *esp_task_wdt_isr_user_handler* in the user code, in order to receive the timeout event and extend the default behavior.

Usage The following functions can be used to watch tasks using the TWDT:

- `esp_task_wdt_init()` to initialize the TWDT and subscribe the idle tasks.
- `esp_task_wdt_add()` subscribes other tasks to the TWDT.
- Once subscribed, `esp_task_wdt_reset()` should be called from the task to feed the TWDT.
- `esp_task_wdt_delete()` unsubscribes a previously subscribed task
- `esp_task_wdt_deinit()` unsubscribes the idle tasks and deinitializes the TWDT

In the case where applications need to watch at a more granular level (i.e., ensure that a particular functions/stub/code-path is called), the TWDT allows subscription of "users".

- `esp_task_wdt_add_user()` to subscribe an arbitrary user of the TWDT. This function will return a user handle to the added user.
- `esp_task_wdt_reset_user()` must be called using the user handle in order to prevent a TWDT timeout.
- `esp_task_wdt_delete_user()` unsubscribes an arbitrary user of the TWDT.

Configuration The default timeout period for the TWDT is set using config item *CONFIG_ESP_TASK_WDT_TIMEOUT_S*. This should be set to at least as long as you expect any single task will need to monopolize the CPU (for example, if you expect the app will do a long intensive calculation and should not yield to other tasks). It is also possible to change this timeout at runtime by calling `esp_task_wdt_init()`.

Note: Erasing large flash areas can be time consuming and can cause a task to run continuously, thus triggering a TWDT timeout. The following two methods can be used to avoid this:

- Increase *CONFIG_ESP_TASK_WDT_TIMEOUT_S* in menuconfig for a larger watchdog timeout period.

- You can also call `esp_task_wdt_init()` to increase the watchdog timeout period before erasing a large flash area.

For more information, you can refer to *SPI Flash*.

The following config options control TWDT configuration. They are all enabled by default:

- *CONFIG_ESP_TASK_WDT_EN* - enables TWDT feature. If this option is disabled, TWDT cannot be used, even if initialized at runtime.
- *CONFIG_ESP_TASK_WDT_INIT* - the TWDT is initialized automatically during startup. If this option is disabled, it is still possible to initialize the Task WDT at runtime by calling `esp_task_wdt_init()`.
- *CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0* - CPU0 Idle task is subscribed to the TWDT during startup. If this option is disabled, it is still possible to subscribe the idle task by calling `esp_task_wdt_init()` again.
- *CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1* - CPU1 Idle task is subscribed to the TWDT during startup.

JTAG & Watchdogs

While debugging using OpenOCD, the CPUs will be halted every time a breakpoint is reached. However if the watchdog timers continue to run when a breakpoint is encountered, they will eventually trigger a reset making it very difficult to debug code. Therefore OpenOCD will disable the hardware timers of both the interrupt and task watchdogs at every breakpoint. Moreover, OpenOCD will not reenable them upon leaving the breakpoint. This means that interrupt watchdog and task watchdog functionality will essentially be disabled. No warnings or panics from either watchdogs will be generated when the ESP32 is connected to OpenOCD via JTAG.

API Reference

Task Watchdog A full example using the Task Watchdog is available in esp-idf: system/task_watchdog

Header File

- components/esp_system/include/esp_task_wdt.h

Functions

esp_err_t **esp_task_wdt_init** (const *esp_task_wdt_config_t* *config)

Initialize the Task Watchdog Timer (TWDT)

This function configures and initializes the TWDT. This function will subscribe the idle tasks if configured to do so. For other tasks, users can subscribe them using esp_task_wdt_add() or esp_task_wdt_add_user(). This function won't start the timer if no task have been registered yet.

Note: esp_task_wdt_init() must only be called after the scheduler is started. Moreover, it must not be called by multiple tasks simultaneously.

Parameters **config** –**[in]** Configuration structure

Returns

- ESP_OK: Initialization was successful
- ESP_ERR_INVALID_STATE: Already initialized
- Other: Failed to initialize TWDT

esp_err_t **esp_task_wdt_reconfigure**(const *esp_task_wdt_config_t* *config)

Reconfigure the Task Watchdog Timer (TWDT)

The function reconfigures the running TWDT. It must already be initialized when this function is called.

Note: esp_task_wdt_reconfigure() must not be called by multiple tasks simultaneously.

Parameters **config** –**[in]** Configuration structure

Returns

- ESP_OK: Reconfiguring was successful
- ESP_ERR_INVALID_STATE: TWDT not initialized yet
- Other: Failed to initialize TWDT

esp_err_t **esp_task_wdt_deinit**(void)

Deinitialize the Task Watchdog Timer (TWDT)

This function will deinitialize the TWDT, and unsubscribe any idle tasks. Calling this function whilst other tasks are still subscribed to the TWDT, or when the TWDT is already deinitialized, will result in an error code being returned.

Note: esp_task_wdt_deinit() must not be called by multiple tasks simultaneously.

Returns

- ESP_OK: TWDT successfully deinitialized
- Other: Failed to deinitialize TWDT

esp_err_t **esp_task_wdt_add**(*TaskHandle_t* task_handle)

Subscribe a task to the Task Watchdog Timer (TWDT)

This function subscribes a task to the TWDT. Each subscribed task must periodically call esp_task_wdt_reset() to prevent the TWDT from elapsing its timeout period. Failure to do so will result in a TWDT timeout.

Parameters **task_handle** –Handle of the task. Input NULL to subscribe the current running task to the TWDT

Returns

- ESP_OK: Successfully subscribed the task to the TWDT
- Other: Failed to subscribe task

esp_err_t **esp_task_wdt_add_user**(const char *user_name, *esp_task_wdt_user_handle_t* *user_handle_ret)

Subscribe a user to the Task Watchdog Timer (TWDT)

This function subscribes a user to the TWDT. A user of the TWDT is usually a function that needs to run periodically. Each subscribed user must periodically call esp_task_wdt_reset_user() to prevent the TWDT from elapsing its timeout period. Failure to do so will result in a TWDT timeout.

Parameters

- **user_name** –**[in]** String to identify the user
- **user_handle_ret** –**[out]** Handle of the user

Returns

- ESP_OK: Successfully subscribed the user to the TWDT
- Other: Failed to subscribe user

esp_err_t **esp_task_wdt_reset**(void)

Reset the Task Watchdog Timer (TWDT) on behalf of the currently running task.

This function will reset the TWDT on behalf of the currently running task. Each subscribed task must periodically call this function to prevent the TWDT from timing out. If one or more subscribed tasks fail to reset the TWDT on their own behalf, a TWDT timeout will occur.

Returns
- ESP_OK: Successfully reset the TWDT on behalf of the currently running task
- Other: Failed to reset

esp_err_t **esp_task_wdt_reset_user**(*esp_task_wdt_user_handle_t* user_handle)

Reset the Task Watchdog Timer (TWDT) on behalf of a user.

This function will reset the TWDT on behalf of a user. Each subscribed user must periodically call this function to prevent the TWDT from timing out. If one or more subscribed users fail to reset the TWDT on their own behalf, a TWDT timeout will occur.

Parameters **user_handle** –**[in]** User handle
- ESP_OK: Successfully reset the TWDT on behalf of the user
- Other: Failed to reset

esp_err_t **esp_task_wdt_delete**(*TaskHandle_t* task_handle)

Unsubscribes a task from the Task Watchdog Timer (TWDT)

This function will unsubscribe a task from the TWDT. After being unsubscribed, the task should no longer call esp_task_wdt_reset().

Parameters **task_handle** –**[in]** Handle of the task. Input NULL to unsubscribe the current running task.

Returns
- ESP_OK: Successfully unsubscribed the task from the TWDT
- Other: Failed to unsubscribe task

esp_err_t **esp_task_wdt_delete_user**(*esp_task_wdt_user_handle_t* user_handle)

Unsubscribes a user from the Task Watchdog Timer (TWDT)

This function will unsubscribe a user from the TWDT. After being unsubscribed, the user should no longer call esp_task_wdt_reset_user().

Parameters **user_handle** –**[in]** User handle

Returns
- ESP_OK: Successfully unsubscribed the user from the TWDT
- Other: Failed to unsubscribe user

esp_err_t **esp_task_wdt_status**(*TaskHandle_t* task_handle)

Query whether a task is subscribed to the Task Watchdog Timer (TWDT)

This function will query whether a task is currently subscribed to the TWDT, or whether the TWDT is initialized.

Parameters **task_handle** –**[in]** Handle of the task. Input NULL to query the current running task.

Returns :
- ESP_OK: The task is currently subscribed to the TWDT
- ESP_ERR_NOT_FOUND: The task is not subscribed
- ESP_ERR_INVALID_STATE: TWDT was never initialized

void **esp_task_wdt_isr_user_handler**(void)

User ISR callback placeholder.

This function is called by task_wdt_isr function (ISR for when TWDT times out). It can be defined in user code to handle TWDT events.

Note: It has the same limitations as the interrupt function. Do not use ESP_LOGx functions inside.

Structures

struct **esp_task_wdt_config_t**

Task Watchdog Timer (TWDT) configuration structure.

Public Members

uint32_t **timeout_ms**

TWDT timeout duration in milliseconds

uint32_t **idle_core_mask**

Mask of the cores who' s idle task should be subscribed on initialization

bool **trigger_panic**

Trigger panic when timeout occurs

Type Definitions

typedef struct esp_task_wdt_user_handle_s ***esp_task_wdt_user_handle_t**

Task Watchdog Timer (TWDT) user handle.

Code examples for this API section are provided in the system directory of ESP-IDF examples.

Chapter 3

Hardware Reference

3.1 Chip Series Comparison

The comparison below covers key features of chips supported by ESP-IDF. For the full list of features please refer to respective datasheets in Section *Related Documents*.

Table 1: Chip Series Comparison

Feature	ESP32 Series	ESP32-S2 Series	ESP32-C3 Series	ESP32-S3 Series
Launch year	2016	2020	2020	2020
Variants	See ESP32 Datasheet (PDF)	See ESP32-S2 Datasheet (PDF)	See ESP32-C3 Datasheet (PDF)	See ESP32-S3 Datasheet (PDF)
Core	Xtensa® dual-/single core 32-bit LX6	Xtensa® single-core 32-bit LX7	32-bit single-core RISC-V	Xtensa® dual-core 32-bit LX7
Wi-Fi protocols	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz
Bluetooth®	Bluetooth v4.2 BR/EDR and Bluetooth Low Energy	×	Bluetooth 5.0	Bluetooth 5.0
Typical frequency	240 MHz (160 MHz for ESP32-S0WD)	240 MHz	160 MHz	240 MHz
SRAM	520 KB	320 KB	400 KB	512 KB
ROM	448 KB for booting and core functions	128 KB for booting and core functions	384 KB for booting and core functions	384 KB for booting and core functions
Embedded flash	2 MB, 4 MB, or none, depending on variants	2 MB, 4 MB, or none, depending on variants	4 MB or none, depending on variants	8 MB or none, depending on variants
External flash	Up to 16 MB device, address 11 MB + 248 KB each time	Up to 1 GB device, address 11.5 MB each time	Up to 16 MB device, address 8 MB each time	Up to 1 GB device, address 32 MB each time
External RAM	Up to 8 MB device, address 4 MB each time	Up to 1 GB device, address 11.5 MB each time	×	Up to 1 GB device, address 32 MB each time

continues on next page

Table 1 – continued from previous page

Feature	ESP32 Series	ESP32-S2 Series	ESP32-C3 Series	ESP32-S3 Series
Cache	✓ Two-way set associative	✓ Four-way set associative, independent instruction cache and data cache	✓ Eight-way set associative, 32-bit data/instruction bus width	✓ Four-way or eight-way set associative for instruction cache; four-way set associative for data cache, 32-bit data/instruction bus width
Peripherals				
ADC	Two 12-bit, 18 channels	Two 12-bit, 20 channels	Two 12-bit SAR ADCs, at most 6 channels	Two 12-bit SAR ADCs, 20 channels
DAC	Two 8-bit channels	Two 8-bit channels	×	×
Timers	Four 64-bit general-purpose timers, and three watchdog timers	Four 64-bit general-purpose timers, and three watchdog timers	Two 54-bit general-purpose timers, and three watchdog timers	Four 54-bit general-purpose timers, and three watchdog timers
Temperature sensor	×	1	1	1
Touch sensor	10	14	×	14
Hall sensor	1	×	×	×
GPIO	34	43	22	45
SPI	4	4	3	4
LCD interface	1	1	×	1
UART	3	2 [1]	2 [1]	3
I2C	2	2	1	2
I2S	2, can be configured to operate with 8/16/32/40/48-bit resolution as an input or output channel.	1, can be configured to operate with 8/16/24/32/48/64-bit resolution as an input or output channel.	1, can be configured to operate with 8/16/24/32-bit resolution as an input or output channel.	2, can be configured to operate with 8/16/24/32-bit resolution as an input or output channel.
Camera interface	1	1	×	1
DMA	Dedicated DMA to UART, SPI, I2S, SDIO slave, SD/MMC host, EMAC, BT, and Wi-Fi	Dedicated DMA to UART, SPI, AES, SHA, I2S, and ADC Controller	General-purpose, 3 TX channels, 3 RX channels	General-purpose, 5 TX channels, 5 RX channels
RMT	8 channels	4 channels [1], can be configured to TX/RX channels	4 channels [2], 2 TX channels, 2 RX channels	8 channels [2], 4 TX channels, 4 RX channels
Pulse counter	8 channels	4 channels [1]	×	4 channels [1]
LED PWM	16 channels	8 channels [1]	6 channels [2]	8 channels [1]
MCPWM	2, six PWM outputs	×	×	2, six PWM outputs
USB OTG	×	1	×	1

continues on next page

Table 1 – continued from previous page

Feature	ESP32 Series	ESP32-S2 Series	ESP32-C3 Series	ESP32-S3 Series
TWAI® controller (compatible with ISO 11898-1)	1	1	1	1
SD/SDIO/MMC host controller	1	×	×	1
SDIO slave controller	1	×	×	×
Ethernet MAC	1	×	×	×
ULP	ULP FSM	PicoRV32 core with 8 KB SRAM, ULP FSM	×	PicoRV32 core with 8 KB SRAM, ULP FSM
Debug Assist	×	×	1	×
Security				
Secure boot	✓	✓ Faster and safer, compared with ESP32	✓ Faster and safer, compared with ESP32	✓ Faster and safer, compared with ESP32
Flash encryption	✓	✓ Support for PSRAM encryption. Safer, compared with ESP32	✓ Safer, compared with ESP32	✓ Support for PSRAM encryption. Safer, compared with ESP32
OTP	1024-bit	4096-bit	4096-bit	4096-bit
AES	✓ AES-128, AES-192, AES-256 (FIPS PUB 197)	✓ AES-128, AES-192, AES-256 (FIPS PUB 197); DMA support	✓ AES-128, AES-256 (FIPS PUB 197); DMA support	✓ AES-128, AES-256 (FIPS PUB 197); DMA support
HASH	SHA-1, SHA-256, SHA-384, SHA-512 (FIPS PUB 180-4)	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, SHA-512/t (FIPS PUB 180-4); DMA support	SHA-1, SHA-224, SHA-256 (FIPS PUB 180-4); DMA support	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, SHA-512/t (FIPS PUB 180-4); DMA support
RSA	Up to 4096 bits	Up to 4096 bits	Up to 3072 bits	Up to 4096 bits
RNG	✓	✓	✓	✓
HMAC	×	✓	✓	✓
Digital signature	×	✓	✓	✓
XTS	×	✓ XTS-AES-128, XTS-AES-256	✓ XTS-AES-128	✓ XTS-AES-128, XTS-AES-256
Other				
Deep-sleep (ULP sensor-monitored pattern)	100 µA (when ADC work with a duty cycle of 1%)	22 µA (when touch sensors work with a duty cycle of 1%)	No such pattern	TBD
Size	QFN48 5*5, 6*6, depending on variants	QFN56 7*7	QFN32 5*5	QFN56 7*7

- **Note** 1: Reduced chip area compared with ESP32
- **Note** 2: Reduced chip area compared with ESP32 and ESP32-S2

- **Note** 3: Die size: ESP32-C3 < ESP32-S2 < ESP32-S3 < ESP32

3.1.1 Related Documents

- ESP32 Datasheet (PDF)
- ESP32-PICO Datasheets (PDF)
  - ESP32-PICO-D4
  - ESP32-PICO-V3
  - ESP32-PICO-V3-02
- ESP32-S2 Datasheet (PDF)
- ESP32-C3 Datasheet (PDF)
- ESP32-S3 Datasheet (PDF)
- ESP Product Selector

Chapter 4

API Guides

4.1 Application Level Tracing library

4.1.1 Overview

ESP-IDF provides a useful feature for program behavior analysis: application level tracing. It is implemented in the corresponding library and can be enabled in menuconfig. This feature allows to transfer arbitrary data between host and ESP32 via JTAG, UART, or USB interfaces with small overhead on program execution. It is possible to use JTAG and UART interfaces simultaneously. The UART interface is mostly used for connection with SEGGER SystemView tool (see SystemView).

Developers can use this library to send application-specific state of execution to the host and receive commands or other types of information from the opposite direction at runtime. The main use cases of this library are:

1. Collecting application-specific data. See *Application Specific Tracing*.
2. Lightweight logging to the host. See *Logging to Host*.
3. System behavior analysis. See *System Behavior Analysis with SEGGER SystemView*.
4. Source code coverage. See *Gcov (Source Code Coverage)*.

Tracing components used when working over JTAG interface are shown in the figure below.

4.1.2 Modes of Operation

The library supports two modes of operation:

Post-mortem mode: This is the default mode. The mode does not need interaction with the host side. In this mode, tracing module does not check whether the host has read all the data from *HW UP BUFFER*, but directly overwrites old data with the new ones. This mode is useful when only the latest trace data is interesting to the user, e.g., for analyzing program ' s behavior just before the crash. The host can read the data later on upon user request, e.g., via special OpenOCD command in case of working via JTAG interface.

Streaming mode: Tracing module enters this mode when the host connects to ESP32. In this mode, before writing new data to *HW UP BUFFER*, the tracing module checks that whether there is enough space in it and if necessary, waits for the host to read data and free enough memory. Maximum waiting time is controlled via timeout values passed by users to corresponding API routines. So when application tries to write data to the trace buffer using the finite value of the maximum waiting time, it is possible that this data will be dropped. This is especially true for tracing from time critical code (ISRs, OS scheduler code, etc.) where infinite timeouts can lead to system malfunction. In order to avoid loss of such critical data, developers can enable additional data buffering via menuconfig option

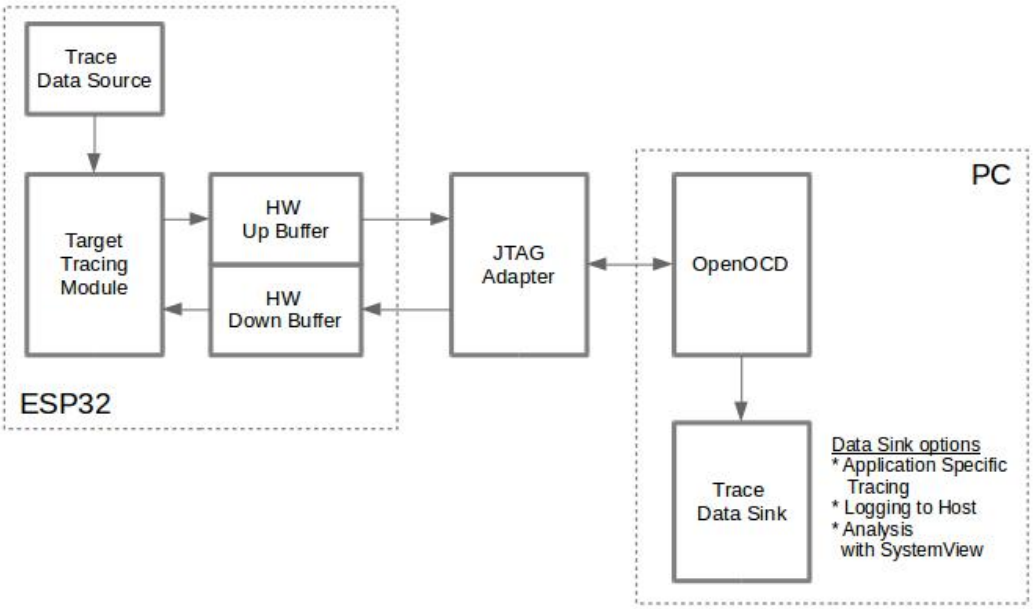


Fig. 1: Tracing Components Used When Working Over JTAG

CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX. This macro specifies the size of data which can be buffered in above conditions. The option can also help to overcome situation when data transfer to the host is temporarily slowed down, e.g., due to USB bus congestions. But it will not help when the average bitrate of the trace data stream exceeds the hardware interface capabilities.

4.1.3 Configuration Options and Dependencies

Using of this feature depends on two components:

1. **Host side:** Application tracing is done over JTAG, so it needs OpenOCD to be set up and running on host machine. For instructions on how to set it up, please see *JTAG Debugging* for details.
2. **Target side:** Application tracing functionality can be enabled in menuconfig. Please go to `Component config` > `Application Level Tracing` menu, which allows selecting destination for the trace data (hardware interface for transport: JTAG or/and UART). Choosing any of the destinations automatically enables the `CONFIG_APPTRACE_ENABLE` option. For UART interfaces, users have to define baud rate, TX and RX pins numbers, and additional UART-related parameters.

Note: In order to achieve higher data rates and minimize the number of dropped packets, it is recommended to optimize the setting of JTAG clock frequency, so that it is at maximum and still provides stable operation of JTAG. See *Optimize JTAG speed*.

There are two additional menuconfig options not mentioned above:

1. *Threshold for flushing last trace data to host on panic* (*CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH*). This option is necessary due to the nature of working over JTAG. In this mode, trace data is exposed to the host in 16 KB blocks. In post-mortem mode, when one block is filled, it is exposed to the host and the previous one becomes unavailable. In other words, the trace data is overwritten in 16 KB granularity. On panic, the latest data from the current input block is exposed to the host and the host can read them for post-analysis. System panic may occur when a very small amount of data are not exposed to the host yet. In this case, the previous 16 KB of collected data will be lost and the host will see the latest, but very small piece of the trace.

It can be insufficient to diagnose the problem. This menuconfig option allows avoiding such situations. It controls the threshold for flushing data in case of apanic. For example, users can decide that it needs no less than 512 bytes of the recent trace data, so if there is less then 512 bytes of pending data at the moment of panic, they will not be flushed and will not overwrite the previous 16 KB. The option is only meaningful in post-mortem mode and when working over JTAG.

2. *Timeout for flushing last trace data to host on panic* (*CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO*). The option is only meaningful in streaming mode and it controls the maximum time that the tracing module will wait for the host to read the last data in case of panic.
3. *UART RX/TX ring buffer size* (*CONFIG_APPTRACE_UART_TX_BUFF_SIZE*). The size of the buffer depends on the amount of data transfered through the UART.
4. *UART TX message size* (*CONFIG_APPTRACE_UART_TX_MSG_SIZE*). The maximum size of the single message to transfer.

4.1.4 How to Use This Library

This library provides APIs for transferring arbitrary data between the host and ESP32. When enabled in menuconfig, the target application tracing module is initialized automatically at the system startup, so all what the user needs to do is to call corresponding APIs to send, receive or flush the data.

Application Specific Tracing

In general, users should decide what type of data should be transferred in every direction and how these data must be interpreted (processed). The following steps must be performed to transfer data between the target and the host:

1. On the target side, users should implement algorithms for writing trace data to the host. Piece of code below shows an example on how to do this.

```
#include "esp_app_trace.h"
...
char buf[] = "Hello World!";
esp_err_t res = esp_apptrace_write(ESP_APPTRACE_DEST_TRAX, buf,␣
↪strlen(buf), ESP_APPTRACE_TMO_INFINITE);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to write data to host!");
    return res;
}
```

`esp_apptrace_write()` function uses memcpy to copy user data to the internal buffer. In some cases, it can be more optimal to use `esp_apptrace_buffer_get()` and `esp_apptrace_buffer_put()` functions. They allow developers to allocate buffer and fill it themselves. The following piece of code shows how to do this.

```
#include "esp_app_trace.h"
...
int number = 10;
char *ptr = (char *)esp_apptrace_buffer_get(ESP_APPTRACE_DEST_TRAX, 32,
↪ 100/*tmo in us*/);
if (ptr == NULL) {
    ESP_LOGE(TAG, "Failed to get buffer!");
    return ESP_FAIL;
}
sprintf(ptr, "Here is the number %d", number);
esp_err_t res = esp_apptrace_buffer_put(ESP_APPTRACE_DEST_TRAX, ptr,␣
↪100/*tmo in us*/);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g., OpenOCD) will report␣
↪incomplete user buffer */
    ESP_LOGE(TAG, "Failed to put buffer!");
    return res;
}
```

Also according to his needs, the user may want to receive data from the host. Piece of code below shows an example on how to do this.

```
#include "esp_app_trace.h"
...
char buf[32];
char down_buf[32];
size_t sz = sizeof(buf);

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
/* check for incoming data and read them if any */
esp_err_t res = esp_apptrace_read(ESP_APPTRACE_DEST_TRAX, buf, &sz, 0/
↪*do not wait*/);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to read data from host!");
    return res;
}
if (sz > 0) {
    /* we have data, process them */
    ...
}
```

`esp_apptrace_read()` function uses memcpy to copy host data to user buffer. In some casesm it can be more optimal to use `esp_apptrace_down_buffer_get()` and `esp_apptrace_down_buffer_put()` functions. They allow developers to occupy chunk of read buffer and process it in-place. The following piece of code shows how to do this.

```
#include "esp_app_trace.h"
...
char down_buf[32];
uint32_t *number;
size_t sz = 32;

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
char *ptr = (char *)esp_apptrace_down_buffer_get(ESP_APPTRACE_DEST_
↪TRAX, &sz, 100/*tmo in us*/);
if (ptr == NULL) {
    ESP_LOGE(TAG, "Failed to get buffer!");
    return ESP_FAIL;
}
if (sz > 4) {
    number = (uint32_t *)ptr;
    printf("Here is the number %d", *number);
} else {
    printf("No data");
}
esp_err_t res = esp_apptrace_down_buffer_put(ESP_APPTRACE_DEST_TRAX,␣
↪ptr, 100/*tmo in us*/);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g., OpenOCD) will report␣
↪incomplete user buffer */
    ESP_LOGE(TAG, "Failed to put buffer!");
    return res;
}
```

2. The next step is to build the program image and download it to the target as described in the *Getting Started Guide*.
3. Run OpenOCD (see *JTAG Debugging*).
4. Connect to OpenOCD telnet server. It can be done using the following command in terminal `telnet <oocd_host> 4444`. If telnet session is opened on the same machine which runs OpenOCD, you can use `localhost` as `<oocd_host>` in the command above.

5. Start trace data collection using special OpenOCD command. This command will transfer tracing data and redirect them to the specified file or socket (currently only files are supported as trace data destination). For description of the corresponding commands, see *OpenOCD Application Level Tracing Commands*.
6. The final step is to process received data. Since the format of data is defined by users, the processing stage is out of the scope of this document. Good starting points for data processor are python scripts in `$IDF_PATH/tools/esp_app_trace`: `apptrace_proc.py` (used for feature tests) and `logtrace_proc.py` (see more details in section *Logging to Host*).

OpenOCD Application Level Tracing Commands *HW UP BUFFER* is shared between user data blocks and the filling of the allocated memory is performed on behalf of the API caller (in task or ISR context). In multithreading environment, it can happen that the task/ISR which fills the buffer is preempted by another high priority task/ISR. So it is possible that the user data preparation process is not completed at the moment when that chunk is read by the host. To handle such conditions, the tracing module prepends all user data chunks with header which contains the allocated user buffer size (2 bytes) and the length of the actually written data (2 bytes). So the total length of the header is 4 bytes. OpenOCD command which reads trace data reports error when it reads incomplete user data chunk, but in any case, it puts the contents of the whole user chunk (including unfilled area) to the output file.

Below is the description of available OpenOCD application tracing commands.

Note: Currently, OpenOCD does not provide commands to send arbitrary user data to the target.

Command usage:

```
esp apptrace [start <options>] | [stop] | [status] | [dump <cores_num> <outfile>]
```

Sub-commands:

`start` Start tracing (continuous streaming).
`stop` Stop tracing.
`status` Get tracing status.
`dump` Dump all data from (post-mortem dump).

Start command syntax:

```
start <outfile> [poll_period [trace_size [stop_tmo [wait4halt [skip_size]]]]]
```

`outfile` Path to file to save data from both CPUs. This argument should have the following format: `file://path/to/file`.
`poll_period` Data polling period (in ms) for available trace data. If greater than 0, then command runs in non-blocking mode. By default, 1 ms.
`trace_size` Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default, -1 (trace size stop trigger is disabled).
`stop_tmo` Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default, -1 (disable this stop trigger). Optionally set it to value longer than longest pause between tracing commands from target.
`wait4halt` If 0, start tracing immediately, otherwise command waits for the target to be halted (after reset, by breakpoint etc.) and then automatically resumes it and starts tracing. By default, 0.
`skip_size` Number of bytes to skip at the start. By default, 0.

Note: If `poll_period` is 0, OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point, tracing stops automatically.

Command usage examples:

1. Collect 2048 bytes of tracing data to the file `trace.log`. The file will be saved in the `openocd-esp32` directory.

   ```
   esp apptrace start file://trace.log 1 2048 5 0 0
   ```

   The tracing data will be retrieved and saved in non-blocking mode. This process will stop automatically after 2048 bytes are collected, or if no data are available for more than 5 seconds.

   Note: Tracing data is buffered before it is made available to OpenOCD. If you see "Data timeout!" message, then it is likely that the target is not sending enough data to empty the buffer to OpenOCD before the timeout. Either increase the timeout or use the function `esp_apptrace_flush()` to flush the data on specific intervals.

2. Retrieve tracing data indefinitely in non-blocking mode.

   ```
   esp apptrace start file://trace.log 1 -1 -1 0 0
   ```

   There is no limitation on the size of collected data and there is no data timeout set. This process may be stopped by issuing `esp apptrace stop` command on OpenOCD telnet prompt, or by pressing Ctrl+C in OpenOCD window.
3. Retrieve tracing data and save them indefinitely.

   ```
   esp apptrace start file://trace.log 0 -1 -1 0 0
   ```

   OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing, press Ctrl+C in the OpenOCD window.
4. Wait for the target to be halted. Then resume the target's operation and start data retrieval. Stop after collecting 2048 bytes of data:

   ```
   esp apptrace start file://trace.log 0 2048 -1 1 0
   ```

   To configure tracing immediately after reset, use the OpenOCD `reset halt` command.

Logging to Host

ESP-IDF implements a useful feature: logging to the host via application level tracing library. This is a kind of semihosting when all *ESP_LOGx* calls send strings to be printed to the host instead of UART. This can be useful because "printing to host" eliminates some steps performed when logging to UART. Most part of the work is done on the host.

By default, ESP-IDF's logging library uses vprintf-like function to write formatted output to dedicated UART. In general, it involves the following steps:

1. Format string is parsed to obtain type of each argument.
2. According to its type, every argument is converted to string representation.
3. Format string combined with converted arguments is sent to UART.

Though the implementation of the vprintf-like function can be optimized to a certain level, all steps above have to be performed in any case and every step takes some time (especially item 3). So it frequently occurs that with additional log added to the program to identify the problem, the program behavior is changed and the problem cannot be reproduced. And in the worst cases, the program cannot work normally at all and ends up with an error or even hangs.

Possible ways to overcome this problem are to use higher UART bitrates (or another faster interface) and/or to move string formatting procedure to the host.

The application level tracing feature can be used to transfer log information to the host using `esp_apptrace_vprintf` function. This function does not perform full parsing of the format string and arguments. Instead, it just calculates the number of arguments passed and sends them along with the format string address to the host. On the host, log data is processed and printed out by a special Python script.

Limitations Current implementation of logging over JTAG has some limitations:

1. No support for tracing from `ESP_EARLY_LOGx` macros.
2. No support for printf arguments whose size exceeds 4 bytes (e.g., `double` and `uint64_t`).
3. Only strings from the .rodata section are supported as format strings and arguments.
4. The maximum number of printf arguments is 256.

How To Use It In order to use logging via trace module, users need to perform the following steps:

1. On the target side, the special vprintf-like function `esp_apptrace_vprintf` needs to be installed. It sends log data to the host. Example code is provided in system/app_trace_to_host.
2. Follow instructions in items 2-5 in *Application Specific Tracing*.
3. To print out collected log records, run the following command in terminal: `$IDF_PATH/tools/esp_app_trace/logtrace_proc.py /path/to/trace/file /path/to/program/elf/file`.

Log Trace Processor Command Options Command usage:

```
logtrace_proc.py [-h] [--no-errors] <trace_file> <elf_file>
```

Positional arguments:

`trace_file` Path to log trace file.
`elf_file` Path to program ELF file.

Optional arguments:

`-h, --help` Show this help message and exit.
`--no-errors, -n` Do not print errors.

System Behavior Analysis with SEGGER SystemView

Another useful ESP-IDF feature built on top of application tracing library is the system level tracing which produces traces compatible with SEGGER SystemView tool (see SystemView). SEGGER SystemView is a real-time recording and visualization tool that allows to analyze runtime behavior of an application. It is possible to view events in real-time through the UART interface.

How To Use It Support for this feature is enabled by `Component config > Application Level Tracing > FreeRTOS SystemView Tracing` (*CONFIG_APPTRACE_SV_ENABLE*) menuconfig option. There are several other options enabled under the same menu:

1. SytemView destination. Select the destination interface: JTAG or UART. In case of UART, it will be possible to connect SystemView application to the ESP32 directly and receive data in real-time.
2. ESP32 timer to use as SystemView timestamp source: (*CONFIG_APPTRACE_SV_TS_SOURCE*) selects the source of timestamps for SystemView events. In the single core mode, timestamps are generated using ESP32 internal cycle counter running at maximum 240 Mhz (~4 ns granularity). In the dual-core mode, external timer working at 40 Mhz is used, so the timestamp granularity is 25 ns.
3. Individually enabled or disabled collection of SystemView events (`CONFIG_APPTRACE_SV_EVT_XXX`):
   - Trace Buffer Overflow Event
   - ISR Enter Event
   - ISR Exit Event
   - ISR Exit to Scheduler Event
   - Task Start Execution Event
   - Task Stop Execution Event
   - Task Start Ready State Event
   - Task Stop Ready State Event
   - Task Create Event
   - Task Terminate Event
   - System Idle Event
   - Timer Enter Event
   - Timer Exit Event

ESP-IDF has all the code required to produce SystemView compatible traces, so users can just configure necessary project options (see above), build, download the image to target, and use OpenOCD to collect data as described in the previous sections.

4. Select Pro or App CPU in menuconfig options `Component config` > `Application Level Tracing` > `FreeRTOS SystemView Tracing` to trace over the UART interface in real-time.

OpenOCD SystemView Tracing Command Options Command usage:

```
esp sysview [start <options>] | [stop] | [status]
```

Sub-commands:

`start` Start tracing (continuous streaming).
`stop` Stop tracing.
`status` Get tracing status.

Start command syntax:

```
start <outfile1> [outfile2] [poll_period [trace_size [stop_tmo]]]
```

`outfile1` Path to file to save data from PRO CPU. This argument should have the following format: `file://path/to/file`.
`outfile2` Path to file to save data from APP CPU. This argument should have the following format: `file://path/to/file`.
`poll_period` Data polling period (in ms) for available trace data. If greater than 0, then command runs in non-blocking mode. By default, 1 ms.
`trace_size` Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default, -1 (trace size stop trigger is disabled).
`stop_tmo` Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default, -1 (disable this stop trigger).

Note: If `poll_period` is 0, OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in the OpenOCD window (not the one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point, tracing stops automatically.

Command usage examples:

1. Collect SystemView tracing data to files `pro-cpu.SVDat` and `app-cpu.SVDat`. The files will be saved in `openocd-esp32` directory.

   ```
   esp sysview start file://pro-cpu.SVDat file://app-cpu.SVDat
   ```

   The tracing data will be retrieved and saved in non-blocking mode. To stop this process, enter `esp sysview stop` command on OpenOCD telnet prompt, optionally pressing Ctrl+C in the OpenOCD window.
2. Retrieve tracing data and save them indefinitely.

   ```
   esp sysview start file://pro-cpu.SVDat file://app-cpu.SVDat 0 -1 -1
   ```

   OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing, press Ctrl+C in the OpenOCD window.

Data Visualization After trace data are collected, users can use a special tool to visualize the results and inspect behavior of the program.

Unfortunately, SystemView does not support tracing from multiple cores. So when tracing from ESP32 with JTAG interfaces in the dual-core mode, two files are generated: one for PRO CPU and another for APP CPU. Users can load each file into separate instances of the tool. For tracing over UART, users can select `Component config` > `Application Level Tracing` > `FreeRTOS SystemView Tracing` in menuconfig Pro or App to choose which CPU has to be traced.

It is uneasy and awkward to analyze data for every core in separate instance of the tool. Fortunately, there is an Eclipse plugin called *Impulse* which can load several trace files, thus making it possible to inspect events from both cores in one view. Also, this plugin has no limitation of 1,000,000 events as compared to the free version of SystemView.

Good instructions on how to install, configure, and visualize data in Impulse from one core can be found here.

Note: ESP-IDF uses its own mapping for SystemView FreeRTOS events IDs, so users need to replace the original file mapping `$SYSVIEW_INSTALL_DIR/Description/SYSVIEW_FreeRTOS.txt` with `$IDF_PATH/tools/esp_app_trace/SYSVIEW_FreeRTOS.txt`. Also, contents of that IDF-specific file should be used when configuring SystemView serializer using the above link.

Configure Impulse for Dual Core Traces After installing Impulse and ensuring that it can successfully load trace files for each core in separate tabs, users can add special Multi Adapter port and load both files into one view. To do this, users need to do the following steps in Eclipse:

1. Open the `Signal Ports` view. Go to `Windows` > `Show View` > `Other menu`. Find the `Signal Ports` view in Impulse folder and double-click it.
2. In the `Signal Ports` view, right-click `Ports` and select `Add` > `New Multi Adapter Port`.
3. In the open dialog box, click `Add` and select `New Pipe/File`.
4. In the open dialog box, select `SystemView Serializer` as Serializer and set path to PRO CPU trace file. Click `OK`.
5. Repeat the steps 3-4 for APP CPU trace file.
6. Double-click the created port. View for this port should open.
7. Click the `Start/Stop Streaming` button. Data should be loaded.
8. Use the `Zoom Out`, `Zoom In` and `Zoom Fit` buttons to inspect data.
9. For settings measurement cursors and other features, please see Impulse documentation).

Note: If you have problems with visualization (no data is shown or strange behaviors of zoom action are observed), you can try to delete current signal hierarchy and double-click on the necessary file or port. Eclipse will ask you to create a new signal hierarchy.

Gcov (Source Code Coverage)

Basics of Gcov and Gcovr Source code coverage is data indicating the count and frequency of every program execution path that has been taken within a program's runtime. Gcov is a GCC tool that, when used in concert with the compiler, can generate log files indicating the execution count of each line of a source file. The Gcovr tool is a utility for managing Gcov and generating summarized code coverage results.

Generally, using Gcov to compile and run programs on the host will undergo these steps:

1. Compile the source code using GCC with the `--coverage` option enabled. This will cause the compiler to generate a `.gcno` notes files during compilation. The notes files contain information to reconstruct execution path block graphs and map each block to source code line numbers. Each source file compiled with the `--coverage` option should have their own `.gcno` file of the same name (e.g., a `main.c` will generate a `main.gcno` when compiled).
2. Execute the program. During execution, the program should generate `.gcda` data files. These data files contain the counts of the number of times an execution path was taken. The program will generate a `.gcda` file for each source file compiled with the `--coverage` option (e.g., `main.c` will generate a `main.gcda`).
3. Gcov or Gcovr can be used to generate a code coverage based on the `.gcno`, `.gcda`, and source files. Gcov will generate a text-based coverage report for each source file in the form of a `.gcov` file, whilst Gcovr will generate a coverage report in HTML format.

Gcov and Gcovr in ESP-IDF Using Gcov in ESP-IDF is complicated due to the fact that the program is running remotely from the host (i.e., on the target). The code coverage data (i.e., the `.gcda` files) is initially stored on the

target itself. OpenOCD is then used to dump the code coverage data from the target to the host via JTAG during runtime. Using Gcov in ESP-IDF can be split into the following steps.

1. *Setting Up a Project for Gcov*
2. *Dumping Code Coverage Data*
3. *Generating Coverage Report*

Setting Up a Project for Gcov

Compiler Option In order to obtain code coverage data in a project, one or more source files within the project must be compiled with the `--coverage` option. In ESP-IDF, this can be achieved at the component level or the individual source file level:

- To cause all source files in a component to be compiled with the `--coverage` option, you can add `target_compile_options(${COMPONENT_LIB} PRIVATE --coverage)` to the `CMakeLists.txt` file of the component.
- To cause a select number of source files (e.g., `source1.c` and `source2.c`) in the same component to be compiled with the `--coverage` option, you can add `set_source_files_properties(source1.c source2.c PROPERTIES COMPILE_FLAGS --coverage)` to the `CMakeLists.txt` file of the component.

When a source file is compiled with the `--coverage` option (e.g., `gcov_example.c`), the compiler will generate the `gcov_example.gcno` file in the project's build directory.

Project Configuration Before building a project with source code coverage, make sure that the following project configuration options are enabled by running `idf.py menuconfig`.

- Enable the application tracing module by selecting `Trace Memory` for the *CONFIG_APPTRACE_DESTINATION1* option.
- Enable Gcov to the host via the *CONFIG_APPTRACE_GCOV_ENABLE*.

Dumping Code Coverage Data Once a project has been complied with the `--coverage` option and flashed onto the target, code coverage data will be stored internally on the target (i.e., in trace memory) whilst the application runs. The process of transferring code coverage data from the target to the host is known as dumping.

The dumping of coverage data is done via OpenOCD (see *JTAG Debugging* on how to setup and run OpenOCD). A dump is triggered by issuing commands to OpenOCD, therefore a telnet session to OpenOCD must be opened to issue such commands (run `telnet localhost 4444`). Note that GDB could be used instead of telnet to issue commands to OpenOCD, however all commands issued from GDB will need to be prefixed as `mon <oocd_command>`.

When the target dumps code coverage data, the `.gcda` files are stored in the project's build directory. For example, if `gcov_example_main.c` of the `main` component is compiled with the `--coverage` option, then dumping the code coverage data would generate a `gcov_example_main.gcda` in `build/esp-idf/main/CMakeFiles/__idf_main.dir/gcov_example_main.c.gcda`. Note that the `.gcno` files produced during compilation are also placed in the same directory.

The dumping of code coverage data can be done multiple times throughout an application's lifetime. Each dump will simply update the `.gcda` file with the newest code coverage information. Code coverage data is accumulative, thus the newest data will contain the total execution count of each code path over the application's entire lifetime.

ESP-IDF supports two methods of dumping code coverage data form the target to the host:

- Instant Run-Time Dumpgit
- Hard-coded Dump

Instant Run-Time Dump An Instant Run-Time Dump is triggered by calling the `ESP32 gcov` OpenOCD command (via a telnet session). Once called, OpenOCD will immediately preempt the ESP32's current state and execute a built-in ESP-IDF Gcov debug stub function. The debug stub function will handle the dumping of data to the host. Upon completion, the ESP32 will resume its current state.

Hard-coded Dump A Hard-coded Dump is triggered by the application itself by calling `esp_gcov_dump()` from somewhere within the application. When called, the application will halt and wait for OpenOCD to connect and retrieve the code coverage data. Once `esp_gcov_dump()` is called, the host must execute the `esp gcov dump` OpenOCD command (via a telnet session). The `esp gcov dump` command will cause OpenOCD to connect to the ESP32, retrieve the code coverage data, then disconnect from the ESP32, thus allowing the application to resume. Hard-coded Dumps can also be triggered multiple times throughout an application's lifetime.

Hard-coded dumps are useful if code coverage data is required at certain points of an application's lifetime by placing `esp_gcov_dump()` where necessary (e.g., after application initialization, during each iteration of an application's main loop).

GDB can be used to set a breakpoint on `esp_gcov_dump()`, then call `mon esp gcov dump` automatically via the use a `gdbinit` script (see Using GDB from *Command Line*).

The following GDB script will add a breakpoint at `esp_gcov_dump()`, then call the `mon esp gcov dump` OpenOCD command.

```
b esp_gcov_dump
commands
mon esp gcov dump
end
```

Note: Note that all OpenOCD commands should be invoked in GDB as: `mon <oocd_command>`.

Generating Coverage Report Once the code coverage data has been dumped, the `.gcno`, `.gcda` and the source files can be used to generate a code coverage report. A code coverage report is simply a report indicating the number of times each line in a source file has been executed.

Both Gcov and Gcovr can be used to generate code coverage reports. Gcov is provided along with the Xtensa toolchain, whilst Gcovr may need to be installed separately. For details on how to use Gcov or Gcovr, refer to Gcov documentation and Gcovr documentation.

Adding Gcovr Build Target to Project To make report generation more convenient, users can define additional build targets in their projects such that the report generation can be done with a single build command.

Add the following lines to the `CMakeLists.txt` file of your project.

```
include($ENV{IDF_PATH}/tools/cmake/gcov.cmake)
idf_create_coverage_report(${CMAKE_CURRENT_BINARY_DIR}/coverage_report)
idf_clean_coverage_report(${CMAKE_CURRENT_BINARY_DIR}/coverage_report)
```

The following commands can now be used:

- `cmake --build build/ --target gcovr-report` will generate an HTML coverage report in `$(BUILD_DIR_BASE)/coverage_report/html` directory.
- `cmake --build build/ --target cov-data-clean` will remove all coverage data files.

4.2 Application Startup Flow

This note explains various steps which happen before `app_main` function of an ESP-IDF application is called.

The high level view of startup process is as follows:

1. *First stage bootloader* in ROM loads second-stage bootloader image to RAM (IRAM & DRAM) from flash offset 0x1000.
2. *Second stage bootloader* loads partition table and main app image from flash. Main app incorporates both RAM segments and read-only segments mapped via flash cache.

3. *Application startup* executes. At this point the second CPU and RTOS scheduler are started.

This process is explained in detail in the following sections.

4.2.1 First stage bootloader

After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located in the mask ROM of the ESP32 chip and cannot be modified.

Startup code called from the reset vector determines the boot mode by checking `GPIO_STRAP_REG` register for bootstrap pin states. Depending on the reset reason, the following takes place:

1. Reset from deep sleep: if the value in `RTC_CNTL_STORE6_REG` is non-zero, and CRC value of RTC memory in `RTC_CNTL_STORE7_REG` is valid, use `RTC_CNTL_STORE6_REG` as an entry point address and jump immediately to it. If `RTC_CNTL_STORE6_REG` is zero, or `RTC_CNTL_STORE7_REG` contains invalid CRC, or once the code called via `RTC_CNTL_STORE6_REG` returns, proceed with boot as if it was a power-on reset. **Note**: to run customized code at this point, a deep sleep stub mechanism is provided. Please see *deep sleep* documentation for this.
2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if a custom boot mode (such as UART Download Mode) is requested. If this is the case, this custom loader mode is executed from ROM. Otherwise, proceed with boot as if it was due to software CPU reset. Consult ESP32 datasheet for a description of SoC boot modes and how to execute them.
3. For software CPU reset and watchdog CPU reset: configure SPI flash based on EFUSE values, and attempt to load the code from flash. This step is described in more detail in the next paragraphs.

Note: During normal boot modes the RTC watchdog is enabled when this happens, so if the process is interrupted or stalled then the watchdog will reset the SOC automatically and repeat the boot process. This may cause the SoC to strap into a new boot mode, if the strapping GPIOs have changed.

Second stage bootloader binary image is loaded from flash starting at address 0x1000. If *Secure Boot* is in use then the first 4 kB sector of flash is used to store secure boot IV and digest of the bootloader image. Otherwise, this sector is unused.

4.2.2 Second stage bootloader

In ESP-IDF, the binary image which resides at offset 0x1000 in flash is the second stage bootloader. Second stage bootloader source code is available in components/bootloader directory of ESP-IDF. Second stage bootloader is used in ESP-IDF to add flexibility to flash layout (using partition tables), and allow for various flows associated with flash encryption, secure boot, and over-the-air updates (OTA) to take place.

When the first stage bootloader is finished checking and loading the second stage bootloader, it jumps to the second stage bootloader entry point found in the binary image header.

Second stage bootloader reads the partition table found by default at offset 0x8000 (*configurable value*). See *partition tables* documentation for more information. The bootloader finds factory and OTA app partitions. If OTA app partitions are found in the partition table, the bootloader consults the `otadata` partition to determine which one should be booted. See *Over The Air Updates (OTA)* for more information.

For a full description of the configuration options available for the ESP-IDF bootloader, see *Bootloader*.

For the selected partition, second stage bootloader reads the binary image from flash one segment at a time:

- For segments with load addresses in internal *IRAM (Instruction RAM)* or *DRAM (Data RAM)*, the contents are copied from flash to the load address.
- For segments which have load addresses in *DROM (data stored in flash)* or *IROM (code executed from flash)* regions, the flash MMU is configured to provide the correct mapping from the flash to the load address.

Note that the second stage bootloader configures flash MMU for both PRO and APP CPUs, but it only enables flash MMU for PRO CPU. Reason for this is that second stage bootloader code is loaded into the memory region used by APP CPU cache. The duty of enabling cache for APP CPU is passed on to the application.

Once all segments are processed - meaning code is loaded and flash MMU is set up, second stage bootloader verifies the integrity of the application and then jumps to the application entry point found in the binary image header.

4.2.3 Application startup

Application startup covers everything that happens after the app starts executing and before the `app_main` function starts running inside the main task. This is split into three stages:

- Port initialization of hardware and basic C runtime environment.
- System initialization of software services and FreeRTOS.
- Running the main task and calling `app_main`.

Note: Understanding all stages of ESP-IDF app initialization is often not necessary. To understand initialization from the application developer's perspective only, skip forward to *Running the main task*.

Port Initialization

ESP-IDF application entry point is `call_start_cpu0` function found in components/esp_system/port/cpu_start.c. This function is executed by the second stage bootloader, and never returns.

This port-layer initialization function initializes the basic C Runtime Environment ("CRT") and performs initial configuration of the SoC's internal hardware:

- Reconfigure CPU exceptions for the app (allowing app interrupt handlers to run, and causing *Fatal Errors* to be handled using the options configured for the app rather than the simpler error handler provided by ROM).
- If the option *CONFIG_BOOTLOADER_WDT_ENABLE* is not set then the RTC watchdog timer is disabled.
- Initialize internal memory (data & bss).
- Finish configuring the MMU cache.
- Enable PSRAM if configured.
- Set the CPU clocks to the frequencies configured for the project.
- Reconfigure the main SPI flash based on the app header settings (necessary for compatibility with bootloader versions before ESP-IDF V4.0, see *Bootloader compatibility*).
- If the app is configured to run on multiple cores, start the other core and wait for it to initialize as well (inside the similar "port layer" initialization function `call_start_cpu1`).

Once `call_start_cpu0` completes running, it calls the "system layer" initialization function `start_cpu0` found in components/esp_system/startup.c. Other cores will also complete port-layer initialization and call `start_other_cores` found in the same file.

System Initialization

The main system initialization function is `start_cpu0`. By default, this function is weak-linked to the function `start_cpu0_default`. This means that it's possible to override this function to add some additional initialization steps.

The primary system initialization stage includes:

- Log information about this application (project name, *App Version*, etc.) if default log level enables this.
- Initialize the heap allocator (before this point all allocations must be static or on the stack).
- Initialize newlib component syscalls and time functions.
- Configure the brownout detector.

- Setup libc stdin, stdout, and stderr according to the *serial console configuration*.
- Perform any security-related checks, including burning efuses that should be burned for this configuration (including *disabling ROM download mode on ESP32 V3*, *CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE*).
- Initialize SPI flash API support.
- Call global C++ constructors and any C functions marked with `__attribute__((constructor))`.

Secondary system initialization allows individual components to be initialized. If a component has an initialization function annotated with the `ESP_SYSTEM_INIT_FN` macro, it will be called as part of secondary initialization. Component initialization functions have priorities assigned to them to ensure the desired initialization order. The priorities are documented in esp_system/system_init_fn.txt and `ESP_SYSTEM_INIT_FN` definition in source code are checked against this file.

Running the main task

After all other components are initialized, the main task is created and the FreeRTOS scheduler starts running.

After doing some more initialization tasks (that require the scheduler to have started), the main task runs the application-provided function `app_main` in the firmware.

The main task that runs `app_main` has a fixed RTOS priority (one higher than the minimum) and a *configurable stack size*.

The main task core affinity is also configurable: *CONFIG_ESP_MAIN_TASK_AFFINITY*.

Unlike normal FreeRTOS tasks (or embedded C `main` functions), the `app_main` task is allowed to return. If this happens, The task is cleaned up and the system will continue running with other RTOS tasks scheduled normally. Therefore, it is possible to implement `app_main` as either a function that creates other application tasks and then returns, or as a main application task itself.

Second core startup

A similar but simpler startup process happens on the APP CPU:

When running system initialization, the code on PRO CPU sets the entry point for APP CPU, de-asserts APP CPU reset, and waits for a global flag to be set by the code running on APP CPU, indicating that it has started. Once this is done, APP CPU jumps to `call_start_cpu1` function in components/esp_system/port/cpu_start.c.

While PRO CPU does initialization in `start_cpu0` function, APP CPU runs `start_cpu_other_cores` function. Similar to `start_cpu0`, this function is weak-linked and defaults to the `start_cpu_other_cores_default` function but can be replaced with a different function by the application.

The `start_cpu_other_cores_default` function does some core-specific system initialization and then waits for the PRO CPU to start the FreeRTOS scheduler, at which point it executes `esp_startup_start_app_other_cores` which is another weak-linked function defaulting to `esp_startup_start_app_other_cores_default`.

By default `esp_startup_start_app_other_cores_default` does nothing but spin in a busy-waiting loop until the scheduler of the PRO CPU triggers an interrupt to start the RTOS scheduler on the APP CPU.

4.3 BluFi

4.3.1 Overview

The BluFi for ESP32 is a Wi-Fi network configuration function via Bluetooth channel. It provides a secure protocol to pass Wi-Fi configuration and credentials to ESP32. Using this information, ESP32 can then connect to an AP or establish a SoftAP.

Fragmenting, data encryption, and checksum verification in the BluFi layer are the key elements of this process.

You can customize symmetric encryption, asymmetric encryption, and checksum support customization. Here we use the DH algorithm for key negotiation, 128-AES algorithm for data encryption, and CRC16 algorithm for checksum verification.

4.3.2 The BluFi Flow

The BluFi networking flow includes the configuration of the SoftAP and Station.

The following uses Station as an example to illustrate the core parts of the procedure, including broadcast, connection, service discovery, negotiation of the shared key, data transmission, and connection status backhaul.

1. Set the ESP32 into GATT Server mode and then it will send broadcasts with specific *advertising data*. You can customize this broadcast as needed, which is not a part of the BluFi Profile.
2. Use the App installed on the mobile phone to search for this particular broadcast. The mobile phone will connect to ESP32 as the GATT Client once the broadcast is confirmed. The App used during this part is up to you.
3. After the GATT connection is successfully established, the mobile phone will send a data frame for key negotiation to ESP32 (see the section *The Frame Formats Defined in BluFi* for details).
4. After ESP32 receives the data frame of key negotiation, it will parse the content according to the user-defined negotiation method.
5. The mobile phone works with ESP32 for key negotiation using the encryption algorithms, such as DH, RSA, or ECC.
6. After the negotiation process is completed, the mobile phone will send a control frame for security-mode setup to ESP32.
7. When receiving this control frame, ESP32 will be able to encrypt and decrypt the communication data using the shared key and the security configuration.
8. The mobile phone sends the data frame defined in the section of *The Frame Formats Defined in BluFi*, with the Wi-Fi configuration information to ESP32, including SSID, password, etc.
9. The mobile phone sends a control frame of Wi-Fi connection request to ESP32. When receiving this control frame, ESP32 will regard the communication of essential information as done and get ready to connect to the Wi-Fi.
10. After connecting to the Wi-Fi, ESP32 will send a control frame of Wi-Fi connection status report to the mobile phone. At this point, the networking procedure is completed.

Note:

1. After ESP32 receives the control frame of security-mode configuration, it will execute the operations in accordance with the defined security mode.
2. The data lengths before and after symmetric encryption/decryption must stay the same. It also supports in-place encryption and decryption.

4.3.3 The Flow Chart of BluFi

4.3.4 The Frame Formats Defined in BluFi

The frame formats for the communication between the mobile phone App and ESP32 are defined as follows:

The frame format with no fragment:

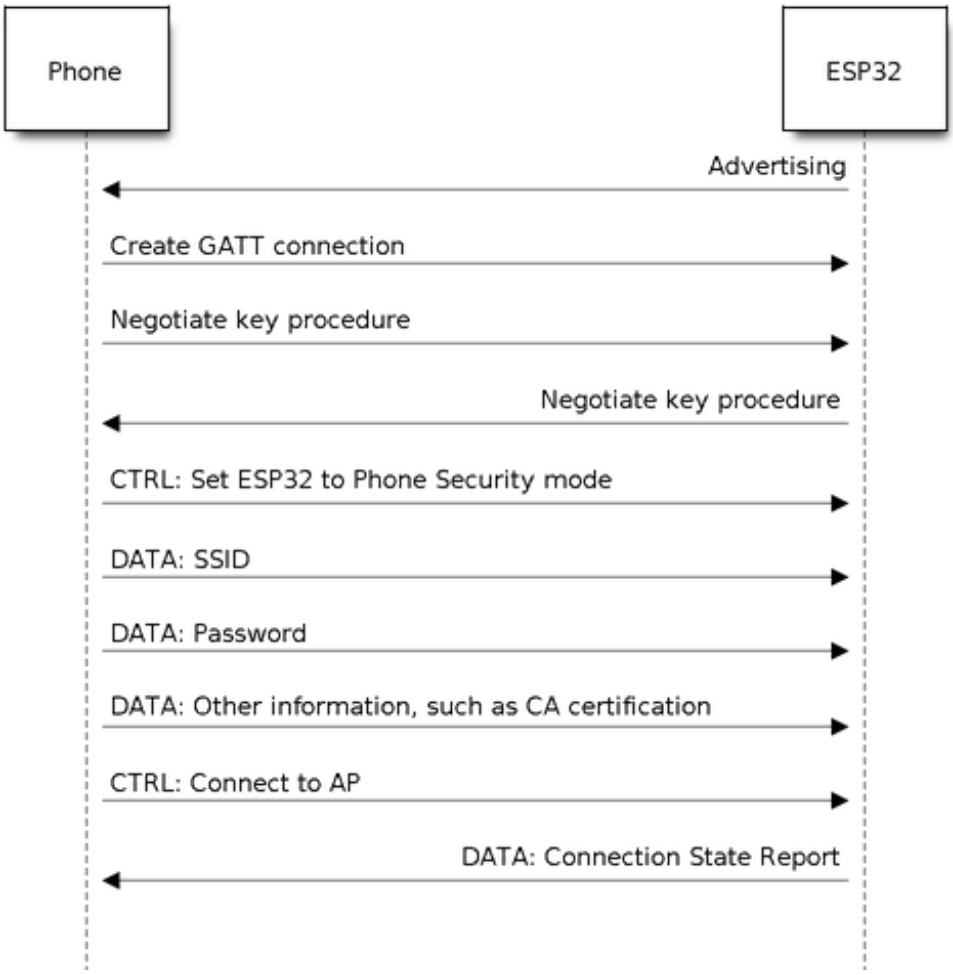


Fig. 2: BluFi Flow Chart

Field	Value (Byte)
Type (Least Significant Bit)	1
Frame Control	1
Sequence Number	1
Data Length	1
Data	${Data Length}
CheckSum (Most Siginificant Bit)	2

If the frag frame bit in the **Frame Control** field is enabled, there would be a 2-byte **Total Content Length** field in the **Data** field. This **Total Content Length** field indicates the length of the remaining part of the frame and also tells the remote how much memory needs to be allocated.

The frame format with fragments:

Field	Value (Byte)
Type (Least Significant Bit)	1
Frame Control (Frag)	1
Sequence Number	1
Data Length	1
Data	• Total Content Length: 2 • Content: ${Data Length} - 2
CheckSum (Most Siginificant Bit)	2

Normally, the control frame does not contain data bits, except for ACK Frame.

The format of ACK Frame:

Field	Value (Byte)
Type - ACK (Least Significant Bit)	1
Frame Control	1
Sequence Number	1
Data Length	1
Data	Acked Sequence Number: 2
CheckSum (Most Siginificant Bit)	2

1. Type
   Type field takes 1 byte and is divided into **Type** and **Subtype**. **Type** uses the lower two bits, indicating whether the frame is a data frame or a control frame. **Subtype** uses the upper six bits, indicating the specific meaning of this data frame or control frame.
   - The control frame is not encrypted for the time being and supports to be verified.
   - The data frame supports to be encrypted and verified.

1.1 Control Frame (Binary: 0x0 b’ 00)

Control Frame	Implication	Explanation	Note
0x0 (b'000000)	ACK	The data field of the ACK frame uses the same sequence value of the frame to reply to.	The data field consumes a byte and its value is the same as the sequence field of the frame to reply to.
0x1 (b'000001)	Set the ESP device to the security mode.	To inform the ESP device of the security mode to use when sending data, which is allowed to be reset multiple times during the process. Each setting affects the subsequent security mode used. If it is not set, the ESP device will send the control frame and data frame with no checksum and encryption by default. The data transmission from the mobile phone to the ESP device is controlled by this control frame.	The data field consumes a byte. The higher four bits are for the security mode setting of the control frame, and the lower four bits are for the security mode setting of the data frame. • b'0000: no checksum and no encryption; • b'0001: with checksum but no encryption; • b'0010: no checksum but with encryption; • b'0011: with both checksum and encryption.
0x2 (b'000010)	Set the opmode of Wi-Fi.	The frame contains opmode settings for configuring the Wi-Fi mode of the ESP device.	data[0] is for opmode settings, including: • 0x00: NULL • 0x01: STA • 0x02: SoftAP • 0x03: SoftAP & STA Please set the SSID/Password/Max Connection Number of the AP mode in the first place if an AP gets involved.
0x3 (b'000011)	Connect the ESP device to the AP.	To notify the ESP device that the essential information has been sent and it is allowed to connect to the AP.	No data field is contained.
0x4 (b'000100)	Disconnect the ESP device from the AP.		No data field is contained.
0x5 (b'000101)	To get the information of the ESP device's Wi-Fi mode and it's status.		• No data field is contained. When receiving this control frame, the ESP device will send back a follow-up frame of Wi-Fi connection state report to the mobile phone with the information of the current opmode, connection status, SSID, and so on. • The types of information sent to the mobile phone is defined by the application installed on the phone.
0x6 (b'000110)	Disconnect the STA device from the SoftAP (in SoftAP mode).		Data[0~5] is taken as the MAC address for the STA device. If there is a second STA device, then it uses data[6-11] and the rest can be done in the same manner.
0x7 (b'000111)	Get the version information.		
0x8 (b'001000)	Disconnect the BLE GATT link.		The ESP device will disconnect the BLE GATT link after receives this command.
0x9 (b'001001)	Get the Wi-Fi list.	To get the ESP device to scan the Wi-Fi access points around.	No data field is contained. When receiving this control frame, the ESP device will send back a follow-up frame of Wi-Fi list report to the mobile phone.

1.2 Data Frame (Binary: 0x1 b’01)

Data Frame	Implication	Explanation	Note
0x0 (b' 000000)	Send the negotiation data.	The negotiation data will be sent to the callback function registered in the application layer.	The length of the data depends on the length field.
0x1 (b' 000001)	Send the SSID for STA mode.	To send the BSSID of the AP for the STA device to connect under the condition that the SSID is hidden.	Please refer to Note 1 below.
0x2 (b' 000010)	Send the SSID for STA mode.	To send the SSID of the AP for the STA device to connect.	Please refer to Note 1 below.
0x3 (b' 000011)	Send the password for STA mode.	To send the password of the AP for the STA device to connect.	Please refer to Note 1 below.
0x4 (b' 000100)	Send the SSID for SoftAP mode.		Please refer to Note 1 below.
0x5 (b' 000101)	Send the password for SoftAPmode.		Please refer to Note 1 below.
0x6 (b' 000110)	Set the maximum connection number for SoftAP mode.		data[0] represents the value of the connection number, ranging from 1 to 4. When the transmission direction is ESP device to the mobile phone, it means to provide the mobile phone with the needed information.
0x7 (b' 000111)	Set the authentication mode for SoftAP mode.		data[0]: • 0x00: OPEN • 0x01: WEP • 0x02: WPA_PSK • 0x03: WPA2_PSK • 0x04: WPA_WPA2_PSK When the transmission direction is from the ESP device to the mobile phone, it means to provide the mobile phone with the needed information.
0x8 (b' 001000)	Set the number of channels for SoftAP mode.		data[0] represents the quantity of the supported channels, ranging from 1 to 14. When the transmission direction is from the ESP device to the mobile phone, it means to provide the mobile phone with the needed information.
0x9 (b' 001001)	Username	It provides the username of the GATT client when using encryption of enterprise level.	The length of the data depends on the length field.
0xa (b' 001010)	CA Certification	It provides the CA Certification when using encryption of enterprise level.	Please refer to Note 2 below.
0xb (b' 001011)	Client Certification	It provides the client certification when using encryption of enterprise level. Whether the private key is contained or not depends on the content of the certification.	Please refer to Note 2 below.
0xc (b' 001100)	Server Certification	It provides the sever certification when using encryption of enterprise level. Whether the private key is contained or not depends on the content of the certification.	Please refer to Note 2 below.
0xd (b' 001101)	Client Private Key	It provides the private key of the client when using encryption of enterprise level.	Please refer to Note 2 below.

Submit Document Feedback

Note:

- Note 1: The length of the data depends on the data length field. When the transmission direction is from the ESP device to the mobile phone, it means to provide the mobile phone with the needed information.
- Note 2: The length of the data depends on the data length field. The frame supports to be fragmented if the data length is not long enough.

2. Frame Control
   The **Frame Control** field takes one byte and each bit has a different meaning.

Bit	Meaning
0x01	Indicates whether the frame is encrypted. • 1 means encrypted. • 0 means unencrypted. The encrypted part of the frame includes the full clear data before the DATA field is encrypted (no checksum). Control frame is not encrypted, so this bit is 0.
0x02	Indicates whether a frame contains a checksum (such as SHA1, MD5, CRC) for the end of the frame. Data field includes sequence, data length, and clear text. Both the control frame and the data frame can choose whether to contain a check bit or not.
0x04	Indicates the data direction. • 0 means from the mobile phone to the ESP device. • 1 means from the ESP device to the mobile phone.
0x08	Indicates whether the other person is required to reply to an ACK. • 0 indicates not required to reply to an ACK. • 1 indicates required to reply to an ACK.
0x10	Indicates whether there are subsequent data fragments. • 0 indicates that there is no subsequent data fragment for this frame. • 1 indicates that there are subsequent data fragments which used to transmit longer data. In the case of a frag frame, the total length of the current content section + subsequent content section is given in the first two bytes of the data field (that is, the content data of the maximum support 64 K).
0x10~0x80	Reserved

3. Sequence Number
   The **Sequence Number** field is the field for sequence control. When a frame is sent, the value of this field is automatically incremented by 1 regardless of the type of frame, which prevents Replay Attack. The sequence would be cleared after each reconnection.
4. Data Length
   The **Data Length** field indicates the length of the data field, which does not include CheckSum.
5. Data
   Content of the **Data** field can be different according to various values of Type or Subtype. Please refer to the table above.
6. CheckSum
   The **CheckSum** field takes two bytes, which is used to check "sequence + data length + clear text data".

4.3.5 The Security Implementation of ESP32

1. Securing Data
   To ensure that the transmission of the Wi-Fi SSID and password is secure, the message needs to be encrypted using symmetric encryption algorithms, such as AES, DES, and so on. Before using symmetric encryption algorithms, the devices are required to negotiate (or generate) a shared key using an asymmetric encryption algorithm (DH, RSA, ECC, etc).
2. Ensuring Data Integrity

To ensure data integrity, you need to add a checksum algorithm, such as SHA1, MD5, CRC, etc.

3. Securing Identity (Signature)
   Algorithm like RSA can be used to secure identity. But for DH, it needs other algorithms as an companion for signature.
4. Replay Attack Prevention
   It is added to the Sequence Number field and used during the checksum verification.
   For the coding of ESP32, you can determine and develop the security processing, such as key negotiation. The mobile application sends the negotiation data to ESP32, and then the data will be sent to the application layer for processing. If the application layer does not process it, you can use the DH encryption algorithm provided by BluFi to negotiate the key.
   The application layer needs to register several security-related functions to BluFi:

```
typedef void (*esp_blufi_negotiate_data_handler_t)(uint8_t *data, int len, uint8_t␣
↪**output_data, int *output_len, bool *need_free)
```

This function is for ESP32 to receive normal data during negotiation. After processing is completed, the data will be transmitted using Output_data and Output_len.

BluFi will send output_data from Negotiate_data_handler after Negotiate_data_handler is called.

Here are two "*" , which means the length of the data to be emitted is unknown. Therefore, it requires the function to allocate itself (malloc) or point to the global variable to inform whether the memory needs to be freed by NEED_FREE.

```
typedef int (* esp_blufi_encrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int␣
↪crypt_len)
```

The data to be encrypted and decrypted must be in the same length. The IV8 is an 8-bit sequence value of frames, which can be used as a 8-bit of IV.

```
typedef int (* esp_blufi_decrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int␣
↪crypt_len)
```

The data to be encrypted and decrypted must be in the same length. The IV8 is an 8-bit sequence value of frames, which can be used as an 8-bit of IV.

```
typedef uint16_t (*esp_blufi_checksum_func_t)(uint8_t iv8, uint8_t *data, int len)
```

This function is used to compute CheckSum and return a value of CheckSum. BluFi uses the returned value to compare the CheckSum of the frame.

4.3.6 GATT Related Instructions

UUID

BluFi Service UUID: 0xFFFF, 16 bit

BluFi (the mobile -> ESP32): 0xFF01, writable

Blufi (ESP32 -> the mobile phone): 0xFF02, readable and callable

4.4 Bootloader

The ESP-IDF Software Bootloader performs the following functions:

1. Minimal initial configuration of internal modules;
2. Initialize *Flash Encryption* and/or *Secure* features, if configured;
3. Select the application partition to boot, based on the partition table and ota_data (if any);

4. Load this image to RAM (IRAM & DRAM) and transfer management to the image that was just loaded.

Bootloader is located at the address 0x1000 in the flash.

For a full description of the startup process including the the ESP-IDF bootloader, see *Application Startup Flow*.

4.4.1 Bootloader compatibility

It is recommended to update to newer *versions of ESP-IDF*: when they are released. The OTA (over the air) update process can flash new apps in the field but cannot flash a new bootloader. For this reason, the bootloader supports booting apps built from newer versions of ESP-IDF.

The bootloader does not support booting apps from older versions of ESP-IDF. When updating ESP-IDF manually on an existing product that might need to downgrade the app to an older version, keep using the older ESP-IDF bootloader binary as well.

Note: If testing an OTA update for an existing product in production, always test it using the same ESP-IDF bootloader binary that is deployed in production.

Before ESP-IDF V2.1

Bootloaders built from very old versions of ESP-IDF (before ESP-IDF V2.1) perform less hardware configuration than newer versions. When using a bootloader from these early ESP-IDF versions and building a new app, enable the config option *CONFIG_APP_COMPATIBLE_PRE_V2_1_BOOTLOADERS*.

Before ESP-IDF V3.1

Bootloaders built from versions of ESP-IDF before V3.1 do not support MD5 checksums in the partition table binary. When using a bootloader from these ESP-IDF versions and building a new app, enable the config option *CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS*.

SPI Flash Configuration

Each ESP-IDF application or bootloader .bin file contains a header with *CONFIG_ESPTOOLPY_FLASHMODE*, *CONFIG_ESPTOOLPY_FLASHFREQ*, *CONFIG_ESPTOOLPY_FLASHSIZE* embedded in it. These are used to configure the SPI flash during boot.

The *First stage bootloader* in ROM reads the *Second stage bootloader* header information from flash and uses this infomation to load the rest of the *Second stage bootloader* from flash. However, at this time the system clock speed is lower than configured and not all flash modes are supported. When the *Second stage bootloader* then runs, it will reconfigure the flash using values read from the currently selected app binary's header (and NOT from the *Second stage bootloader* header). This allows an OTA update to change the SPI flash settings in use.

Bootloaders prior to ESP-IDF V4.0 used the bootloader's own header to configure the SPI flash, meaning these values could not be changed in an update. To maintain compatibility with older bootloaders, the app re-initializes the flash settings during app startup using the configuration found in the app header.

4.4.2 Log Level

The default bootloader log level is "Info". By setting the *CONFIG_BOOTLOADER_LOG_LEVEL* option, it's possible to increase or decrease this level. This log level is separate from the log level used in the app (see *Logging library*).

Reducing bootloader log verbosity can improve the overall project boot time by a small amount.

4.4.3 Factory reset

Sometimes it is desirable to have a way for the device to fall back to a known-good state, in case of some problem with an update.

To roll back to the original "factory" device configuration and clear any user settings, configure the config item *CONFIG_BOOTLOADER_FACTORY_RESET* in the bootloader.

The factory reset mechanism allows the device to be factory reset in two ways:

- Clear one or more data partitions. The *CONFIG_BOOTLOADER_DATA_FACTORY_RESET* option allows users to specify which data partitions will be erased when the factory reset is executed.
  Users can specify the names of partitions as a comma-delimited list with optional spaces for readability. (Like this: `nvs, phy_init, nvs_custom`).
  Make sure that the names of partitions specified in the option are the same as those found in the partition table. Partitions of type "app" cannot be specified here.
- Boot from "factory" app partition. Enabling the *CONFIG_BOOTLOADER_OTA_DATA_ERASE* option will cause the device to boot from the default "factory" app partition after a factory reset (or if there is no factory app partition in the partition table then the default ota app partition is selected instead). This reset process involves erasing the OTA data partition which holds the currently selected OTA partition slot. The "factory" app partition slot (if it exists) is never updated via OTA, so resetting to this allows reverting to a "known good" firmware application.

Either or both of these configuration options can be enabled independently.

In addition, the following configuration options control the reset condition:

- *CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET*- The input GPIO number used to trigger a factory reset. This GPIO must be pulled low or high (configurable) on reset to trigger this.
- *CONFIG_BOOTLOADER_HOLD_TIME_GPIO*- this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.
- *CONFIG_BOOTLOADER_FACTORY_RESET_PIN_LEVEL* - configure whether a factory reset should trigger on a high or low level of the GPIO. If the GPIO has an internal pullup then this is enabled before the pin is sampled, consult the ESP32 datasheet for details on pin internal pullups.

4.4.4 Boot from Test Firmware

It's possible to write a special firmware app for testing in production, and boot this firmware when needed. The project partition table will need a dedicated app partition entry for this testing app, type `app` and subtype `test` (see *Partition Tables*).

Implementing a dedicated test app firmware requires creating a totally separate ESP-IDF project for the test app (each project in ESP-IDF only builds one app). The test app can be developed and tested independently of the main project, and then integrated at production testing time as a pre-compiled .bin file which is flashed to the address of the main project's test app partition.

To support this functionality in the main project's bootloader, set the configuration item *CONFIG_BOOTLOADER_APP_TEST* and configure the following two items:

- *CONFIG_BOOTLOADER_NUM_PIN_APP_TEST* - GPIO number to boot TEST partition. The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset.
  Once the GPIO input is released (allowing it to be pulled up) and the device has been reboot, the normally configured application will boot (factory or any OTA app partition slot).
- *CONFIG_BOOTLOADER_HOLD_TIME_GPIO* - this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

4.4.5 Rollback

Rollback and anti-rollback features must be configured in the bootloader as well.

Consult the *App rollback* and *Anti-rollback* sections in the *OTA API reference document*.

4.4.6 Watchdog

By default, the hardware RTC Watchdog timer remains running while the bootloader is running and will automatically reset the chip if no app has successfully started after 9 seconds.

- The timeout period can be adjusted by setting *CONFIG_BOOTLOADER_WDT_TIME_MS* and recompiling the bootloader.
- The app's behaviour can be adjusted so the RTC Watchdog remains enabled after app startup. The Watchdog would need to be explicitly reset (i.e., fed) by the app to avoid a reset. To do this, set the *CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE* option, modify the app as needed, and then recompile the app.
- The RTC Watchdog can be disabled in the bootloader by disabling the *CONFIG_BOOTLOADER_WDT_ENABLE* setting and recompiling the bootloader. This is not recommended.

4.4.7 Bootloader Size

When enabling additional bootloader functions, including *Flash Encryption* or Secure Boot, and especially if setting a high *CONFIG_BOOTLOADER_LOG_LEVEL* level, then it is important to monitor the bootloader .bin file's size.

When using the default *CONFIG_PARTITION_TABLE_OFFSET* value 0x8000, the size limit is 0x7000 (28672) bytes.

If the bootloader binary is too large, then the bootloader build will fail with an error "Bootloader binary size [..] is too large for partition table offset". If the bootloader binary is flashed anyhow then the ESP32 will fail to boot - errors will be logged about either invalid partition table or invalid bootloader checksum.

Options to work around this are:

- Set *bootloader compiler optimization* back to "Size" if it has been changed from this default value.
- Reduce *bootloader log level*. Setting log level to Warning, Error or None all significantly reduce the final binary size (but may make it harder to debug).
- Set *CONFIG_PARTITION_TABLE_OFFSET* to a higher value than 0x8000, to place the partition table later in the flash. This increases the space available for the bootloader. If the *partition table* CSV file contains explicit partition offsets, they will need changing so no partition has an offset lower than `CONFIG_PARTITION_TABLE_OFFSET + 0x1000`. (This includes the default partition CSV files supplied with ESP-IDF.)

When Secure Boot V2 is enabled, there is also an absolute binary size limit of 48KB (0xC000 bytes) (excluding the 4 KB signature), because the bootloader is first loaded into a fixed size buffer for verification.

4.4.8 Fast boot from Deep Sleep

The bootloader has the *CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP* option which allows the wake-up time from deep sleep to be reduced (useful for reducing power consumption). This option is available when *CONFIG_SECURE_BOOT* option is disabled. Reduction of time is achieved due to the lack of image verification. During the first boot, the bootloader stores the address of the application being launched in the RTC FAST memory. And during the awakening, this address is used for booting without any checks, thus fast loading is achieved.

4.4.9 Custom bootloader

The current bootloader implementation allows a project to extend it or modify it. There are two ways of doing it: by implementing hooks or by overriding it. Both ways are presented in custom_bootloader folder in ESP-IDF examples:

- *bootloader_hooks* which presents how to connect some hooks to the bootloader initialization
- *bootloader_override* which presents how to override the bootloader implementation

In the bootloader space, you cannot use the drivers and functions from other components. If necessary, then the required functionality should be placed in the project's *bootloader_components* directory (note that this will increase its size).

If the bootloader grows too large then it can collide with the partition table, which is flashed at offset 0x8000 by default. Increase the *partition table offset* value to place the partition table later in the flash. This increases the space available for the bootloader.

4.5 Build System

This document explains the implementation of the ESP-IDF build system and the concept of "components". Read this document if you want to know how to organize and build a new ESP-IDF project or component.

4.5.1 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP-IDF base libraries (libc, ROM bindings, etc)
- The Wi-Fi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A webserver
- A driver for the humidity sensor
- Main code tying it all together

ESP-IDF makes these components explicit and configurable. To do that, when a project is compiled, the build system will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a text-based menu system to customize each component. After the components in the project are configured, the build system will compile the project.

Concepts

- A "project" is a directory that contains all the files and configuration to build a single "app" (executable), as well as additional supporting elements such as a partition table, data/filesystem partitions, and a bootloader.
- "Project configuration" is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `idf.py menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An "app" is an executable which is built by ESP-IDF. A single project will usually build two apps - a "project app" (the main executable, ie your custom firmware) and a "bootloader app" (the initial bootloader program which launches the project app).
- "components" are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by ESP-IDF itself, others may be sourced from other places.
- "Target" is the hardware for which an application is built. A full list of supported targets in your version of ESP-IDF can be seen by running *idf.py –list-targets*.

Some things are not part of the project:

- "ESP-IDF" is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.

- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line PATH.

4.5.2 Using the Build System

idf.py

The `idf.py` command-line tool provides a front-end for easily managing your project builds. It manages the following tools:

- CMake, which configures the project to be built
- Ninja which builds the project
- esptool.py for flashing the target.

You can read more about configuring the build system using `idf.py` *here*.

Using CMake Directly

idf.py is a wrapper around CMake for convenience. However, you can also invoke CMake directly if you prefer.

When `idf.py` does something, it prints each command that it runs for easy reference. For example, the `idf.py build` command is the same as running these commands in a bash shell (or similar commands for Windows Command Prompt):

```
mkdir -p build
cd build
cmake .. -G Ninja    # or 'Unix Makefiles'
ninja
```

In the above list, the `cmake` command configures the project and generates build files for use with the final build tool. In this case the final build tool is Ninja: running `ninja` actually builds the project.

It's not necessary to run `cmake` more than once. After the first build, you only need to run `ninja` each time. `ninja` will automatically re-invoke `cmake` if the project needs reconfiguration.

If using CMake with `ninja` or `make`, there are also targets for more of the `idf.py` sub-commands - for example running `make menuconfig` or `ninja menuconfig` in the build directory will work the same as `idf.py menuconfig`.

Note: If you're already familiar with CMake, you may find the ESP-IDF CMake-based build system unusual because it wraps a lot of CMake's functionality to reduce boilerplate. See *writing pure CMake components* for some information about writing more "CMake style" components.

Flashing with ninja or make It's possible to build and flash directly from ninja or make by running a target like:

```
ninja flash
```

Or:

```
make app-flash
```

Available targets are: `flash`, `app-flash` (app only), `bootloader-flash` (bootloader only).

When flashing this way, optionally set the `ESPPORT` and `ESPBAUD` environment variables to specify the serial port and baud rate. You can set environment variables in your operating system or IDE project. Alternatively, set them directly on the command line:

```
ESPPORT=/dev/ttyUSB0 ninja flash
```

Note: Providing environment variables at the start of the command like this is Bash shell Syntax. It will work on Linux and macOS. It won't work when using Windows Command Prompt, but it will work when using Bash-like shells on Windows.

Or:

```
make -j3 app-flash ESPPORT=COM4 ESPBAUD=2000000
```

Note: Providing variables at the end of the command line is `make` syntax, and works for `make` on all platforms.

Using CMake in an IDE

You can also use an IDE with CMake integration. The IDE will want to know the path to the project's `CMakeLists.txt` file. IDEs with CMake integration often provide their own build tools (CMake calls these "generators") to build the source files as part of the IDE.

When adding custom non-build steps like "flash" to the IDE, it is recommended to execute `idf.py` for these "special" commands.

For more detailed information about integrating ESP-IDF with CMake into an IDE, see *Build System Metadata*.

Setting up the Python Interpreter

ESP-IDF works well with Python version 3.7+.

`idf.py` and other Python scripts will run with the default Python interpreter, i.e. `python`. You can switch to a different one like `python3 $IDF_PATH/tools/idf.py ...`, or you can set up a shell alias or another script to simplify the command.

If using CMake directly, running `cmake -D PYTHON=python3 ...` will cause CMake to override the default Python interpreter.

If using an IDE with CMake, setting the `PYTHON` value as a CMake cache override in the IDE UI will override the default Python interpreter.

To manage the Python version more generally via the command line, check out the tools pyenv or virtualenv. These let you change the default Python version.

Possible issues The user of `idf.py` may sometimes experience `ImportError` described below.

```
Traceback (most recent call last):
  File "/Users/user_name/e/esp-idf/tools/kconfig_new/confgen.py", line 27, in
↪<module>
    import kconfiglib
ImportError: bad magic number in 'kconfiglib': b'\x03\xf3\r\n'
```

The exception is often caused by `.pyc` files generated by different Python versions. To solve the issue run the following command:

```
idf.py python-clean
```

4.5.3 Example Project

An example project directory tree might look like this:

```
- myProject/
             - CMakeLists.txt
             - sdkconfig
             - components/ - component1/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                           - component2/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                                         - include/ - component2.h
             - main/       - CMakeLists.txt
                           - src1.c
                           - src2.c

             - build/
```

This example "myProject" contains the following elements:

- A top-level project CMakeLists.txt file. This is the primary file which CMake uses to learn how to build the project; and may set project-wide CMake variables. It includes the file /tools/cmake/project.cmake which implements the rest of the build system. Finally, it sets the project name and defines the project.
- "sdkconfig" project configuration file. This file is created/updated when `idf.py menuconfig` runs, and holds configuration for all of the components in the project (including ESP-IDF itself). The "sdkconfig" file may or may not be added to the source control system of the project.
- Optional "components" directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren't part of ESP-IDF. Alternatively, `EXTRA_COMPONENT_DIRS` can be set in the top-level CMakeLists.txt to look for components in other places.
- "main" directory is a special component that contains source code for the project itself. "main" is a default name, the CMake variable `COMPONENT_DIRS` includes this component but you can modify this variable. See the *renaming main* section for more info. If you have a lot of source files in your project, we recommend grouping most into components instead of putting them all in "main".
- "build" directory is where build output is created. This directory is created by `idf.py` if it doesn't already exist. CMake configures the project and generates interim build files in this directory. Then, after the main build process is run, this directory will also contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories each contain a component `CMakeLists.txt` file. This file contains variable definitions to control the build process of the component, and its integration into the overall project. See *Component CMakeLists Files* for more details.

Each component may also include a `Kconfig` file defining the *component configuration* options that can be set via `menuconfig`. Some components may also include `Kconfig.projbuild` and `project_include.cmake` files, which are special files for *overriding parts of the project*.

4.5.4 Project CMakeLists File

Each project has a single top-level `CMakeLists.txt` file that contains build settings for the entire project. By default, the project CMakeLists can be quite minimal.

Minimal Example CMakeLists

Minimal project:

```
cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(myProject)
```

Mandatory Parts

The inclusion of these three lines, in the order shown above, is necessary for every project:

- `cmake_minimum_required(VERSION 3.16)` tells CMake the minimum version that is required to build the project. ESP-IDF is designed to work with CMake 3.16 or newer. This line must be the first line in the CMakeLists.txt file.
- `include($ENV{IDF_PATH}/tools/cmake/project.cmake)` pulls in the rest of the CMake functionality to configure the project, discover all the components, etc.
- `project(myProject)` creates the project itself, and specifies the project name. The project name is used for the final binary output files of the app - ie `myProject.elf`, `myProject.bin`. Only one project can be defined per CMakeLists file.

Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in /tools/cmake/project.cmake for all of the implementation details.

- `COMPONENT_DIRS`: Directories to search for components. Defaults to `IDF_PATH/components`, `PROJECT_DIR/components`, and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components. Paths can be relative to the project directory, or absolute.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories. Use this variable to "trim down" the project for faster build times. Note that any component which "requires" another component via the REQUIRES or PRIV_REQUIRES arguments on component registration will automatically have it added to this list, so the `COMPONENTS` list can be very short.

Any paths in these variables can be absolute paths, or set relative to the project directory.

To set these variables, use the cmake set command ie `set(VARIABLE "VALUE")`. The `set()` commands should be placed after the `cmake_minimum(...)` line but before the `include(...)` line.

Renaming `main` component

The build system provides special treatment to the `main` component. It is a component that gets automatically added to the build provided that it is in the expected location, PROJECT_DIR/main. All other components in the build are also added as its dependencies, saving the user from hunting down dependencies and providing a build that works right out of the box. Renaming the `main` component causes the loss of these behind-the-scenes heavy lifting, requiring the user to specify the location of the newly renamed component and manually specifying its dependencies. Specifically, the steps to renaming `main` are as follows:

1. Rename `main` directory.
2. Set `EXTRA_COMPONENT_DIRS` in the project CMakeLists.txt to include the renamed `main` directory.
3. Specify the dependencies in the renamed component's CMakeLists.txt file via REQUIRES or PRIV_REQUIRES arguments *on component registration*.

Overriding default build specifications

The build sets some global build specifications (compile flags, definitions, etc.) that gets used in compiling all sources from all components.

For example, one of the default build specifications set is the compile option `-Wextra`. Suppose a user wants to use override this with `-Wno-extra`, it should be done after `project()`:

```
cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(myProject)

idf_build_set_property(COMPILE_OPTIONS "-Wno-error" APPEND)
```

This ensures that the compile options set by the user won't be overriden by the default build specifications, since the latter are set inside `project()`.

4.5.5 Component CMakeLists Files

Each project contains one or more components. Components can be part of ESP-IDF, part of the project's own components directory, or added from custom component directories (*see above*).

A component is any directory in the `COMPONENT_DIRS` list which contains a `CMakeLists.txt` file.

Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a *CMakeLists.txt* file), or they can be top-level directories whose sub-directories are components.

When CMake runs to configure the project, it logs the components included in the build. This list can be useful for debugging the inclusion/exclusion of certain components.

Multiple components with the same name

When ESP-IDF is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means ESP-IDF's internal components first (`IDF_PATH/components`), then any components in directories specified in `EXTRA_COMPONENT_DIRS`, and finally the project's components (`PROJECT_DIR/components`). If two or more of these directories contain component sub-directories with the same name, the component in the last place searched is used. This allows, for example, overriding ESP-IDF components with a modified version by copying that component from the ESP-IDF components directory to the project components directory and then modifying it there. If used in this way, the ESP-IDF directory itself can remain untouched.

Note: If a component is overridden in an existing project by moving it to a new location, the project will not automatically see the new component path. Run `idf.py reconfigure` (or delete the project build folder) and then build again.

Minimal Component CMakeLists

The minimal component `CMakeLists.txt` file simply registers the component to the build system using `idf_component_register`:

```
idf_component_register(SRCS "foo.c" "bar.c"
                       INCLUDE_DIRS "include"
                       REQUIRES mbedtls)
```

- `SRCS` is a list of source files (`*.c`, `*.cpp`, `*.cc`, `*.S`). These source files will be compiled into the component library.

- INCLUDE_DIRS is a list of directories to add to the global include search path for any component which requires this component, and also the main source files.
- REQUIRES is not actually required, but it is very often required to declare what other components this component will use. See *component requirements*.

A library with the name of the component will be built and linked into the final app.

Directories are usually specified relative to the CMakeLists.txt file itself, although they can be absolute.

There are other arguments that can be passed to idf_component_register. These arguments are discussed *here*.

See *example component requirements* and *example component CMakeLists* for more complete component CMakeLists.txt examples.

Preset Component Variables

The following component-specific variables are available for use inside component CMakeLists, but should not be modified:

- COMPONENT_DIR: The component directory. Evaluates to the absolute path of the directory containing CMakeLists.txt. The component path cannot contain spaces. This is the same as the CMAKE_CURRENT_SOURCE_DIR variable.
- COMPONENT_NAME: Name of the component. Same as the name of the component directory.
- COMPONENT_ALIAS: Alias of the library created internally by the build system for the component.
- COMPONENT_LIB: Name of the library created internally by the build system for the component.

The following variables are set at the project level, but available for use in component CMakeLists:

- CONFIG_*: Each value in the project configuration has a corresponding variable available in cmake. All names begin with CONFIG_. *More information here*.
- ESP_PLATFORM: Set to 1 when the CMake file is processed within ESP-IDF build system.

Build/Project Variables

The following are some project/build variables that are available as build properties and whose values can be queried using idf_build_get_property from the component CMakeLists.txt:

- PROJECT_NAME: Name of the project, as set in project CMakeLists.txt file.
- PROJECT_DIR: Absolute path of the project directory containing the project CMakeLists. Same as the CMAKE_SOURCE_DIR variable.
- COMPONENTS: Names of all components that are included in this build, formatted as a semicolon-delimited CMake list.
- IDF_VER: Git version of ESP-IDF (produced by git describe)
- IDF_VERSION_MAJOR, IDF_VERSION_MINOR, IDF_VERSION_PATCH: Components of ESP-IDF version, to be used in conditional expressions. Note that this information is less precise than that provided by IDF_VER variable. v4.0-dev-*, v4.0-beta1, v4.0-rc1 and v4.0 will all have the same values of IDF_VERSION_* variables, but different IDF_VER values.
- IDF_TARGET: Name of the target for which the project is being built.
- PROJECT_VER: Project version.
  - If *CONFIG_APP_PROJECT_VER_FROM_CONFIG* option is set, the value of *CONFIG_APP_PROJECT_VER* will be used.
  - Else, if PROJECT_VER variable is set in project CMakeLists.txt file, its value will be used.
  - Else, if the PROJECT_DIR/version.txt exists, its contents will be used as PROJECT_VER.
  - Else, if the project is located inside a Git repository, the output of git describe will be used.
  - Otherwise, PROJECT_VER will be "1".
- EXTRA_PARTITION_SUBTYPES: CMake list of extra partition subtypes. Each subtype description is a comma separated string with type_name, subtype_name, numeric_value format. Components may add new subtypes by appending them to this list.

Other build properties are listed *here*.

Controlling Component Compilation

To pass compiler options when compiling source files belonging to a particular component, use the target_compile_options function:

```
target_compile_options(${COMPONENT_LIB} PRIVATE -Wno-unused-variable)
```

To apply the compilation flags to a single source file, use the CMake set_source_files_properties command:

```
set_source_files_properties(mysrc.c
    PROPERTIES COMPILE_FLAGS
    -Wno-unused-variable
)
```

This can be useful if there is upstream code that emits warnings.

When using these commands, place them after the call to `idf_component_register` in the component CMakeLists file.

4.5.6 Component Configuration

Each component can also have a `Kconfig` file, alongside `CMakeLists.txt`. This contains configuration settings to add to the configuration menu for this component.

These settings are found under the "Component Settings" menu when menuconfig is run.

To create a component Kconfig file, it is easiest to start with one of the Kconfig files distributed with ESP-IDF.

For an example, see *Adding conditional configuration*.

4.5.7 Preprocessor Definitions

The ESP-IDF build system adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` : Can be used to detect that build happens within ESP-IDF.
- `IDF_VER` : Defined to a git version string. E.g. `v2.0` for a tagged release or `v1.0-275-g0efaa4f` for an arbitrary commit.

4.5.8 Component Requirements

When compiling each component, the ESP-IDF build system recursively evaluates its dependencies. This means each component needs to declare the components that it depends on ("requires").

When writing a component

```
idf_component_register(...
                       REQUIRES mbedtls
                       PRIV_REQUIRES console spiffs)
```

- `REQUIRES` should be set to all components whose header files are #included from the *public* header files of this component.
- `PRIV_REQUIRES` should be set to all components whose header files are #included from *any source files* in this component, unless already listed in `REQUIRES`. Also any component which is required to be linked in order for this component to function correctly.
- The values of `REQUIRES` and `PRIV_REQUIRES` should not depend on any configuration choices (`CONFIG_xxx` macros). This is because requirements are expanded before configuration is loaded. Other component variables (like include paths or source files) can depend on configuration choices.

- Not setting either or both REQUIRES variables is fine. If the component has no requirements except for the *Common component requirements* needed for RTOS, libc, etc.

If a components only supports some target chips (values of IDF_TARGET) then it can specify REQUIRED_IDF_TARGETS in the idf_component_register call to express these requirements. In this case the build system will generate an error if the component is included into the build, but does not support the selected target.

Note: In CMake terms, REQUIRES & PRIV_REQUIRES are approximate wrappers around the CMake functions target_link_libraries(... PUBLIC ...) and target_link_libraries(... PRIVATE ...).

Example of component requirements

Imagine there is a car component, which uses the engine component, which uses the spark_plug component:

```
- autoProject/
             - CMakeLists.txt
             - components/ - car/ - CMakeLists.txt
                                  - car.c
                                  - car.h
                           - engine/ - CMakeLists.txt
                                     - engine.c
                                     - include/ - engine.h
                           - spark_plug/  - CMakeLists.txt
                                          - spark_plug.c
                                          - spark_plug.h
```

Car component The car.h header file is the public interface for the car component. This header includes engine.h directly because it uses some declarations from this header:

```
/* car.h */
#include "engine.h"

#ifdef ENGINE_IS_HYBRID
#define CAR_MODEL "Hybrid"
#endif
```

And car.c includes car.h as well:

```
/* car.c */
#include "car.h"
```

This means the car/CMakeLists.txt file needs to declare that car requires engine:

```
idf_component_register(SRCS "car.c"
                    INCLUDE_DIRS "."
                    REQUIRES engine)
```

- SRCS gives the list of source files in the car component.
- INCLUDE_DIRS gives the list of public include directories for this component. Because the public interface is car.h, the directory containing car.h is listed here.
- REQUIRES gives the list of components required by the public interface of this component. Because car.h is a public header and includes a header from engine, we include engine here. This makes sure that any other component which includes car.h will be able to recursively include the required engine.h also.

Engine component The `engine` component also has a public header file `include/engine.h`, but this header is simpler:

```
/* engine.h */
#define ENGINE_IS_HYBRID

void engine_start(void);
```

The implementation is in `engine.c`:

```
/* engine.c */
#include "engine.h"
#include "spark_plug.h"

...
```

In this component, `engine` depends on `spark_plug` but this is a private dependency. `spark_plug.h` is needed to compile `engine.c`, but not needed to include `engine.h`.

This means that the `engine/CMakeLists.txt` file can use `PRIV_REQUIRES`:

```
idf_component_register(SRCS "engine.c"
                    INCLUDE_DIRS "include"
                    PRIV_REQUIRES spark_plug)
```

As a result, source files in the `car` component don't need the `spark_plug` include directories added to their compiler search path. This can speed up compilation, and stops compiler command lines from becoming longer than necessary.

Spark Plug Component The `spark_plug` component doesn't depend on anything else. It has a public header file `spark_plug.h`, but this doesn't include headers from any other components.

This means that the `spark_plug/CMakeLists.txt` file doesn't need any `REQUIRES` or `PRIV_REQUIRES` clauses:

```
idf_component_register(SRCS "spark_plug.c"
                    INCLUDE_DIRS ".")
```

Source File Include Directories

Each component's source file is compiled with these include path directories, as specified in the passed arguments to `idf_component_register`:

```
idf_component_register(..
                        INCLUDE_DIRS "include"
                        PRIV_INCLUDE_DIRS "other")
```

- The current component's `INCLUDE_DIRS` and `PRIV_INCLUDE_DIRS`.
- The `INCLUDE_DIRS` belonging to all other components listed in the `REQUIRES` and `PRIV_REQUIRES` parameters (ie all the current component's public and private dependencies).
- Recursively, all of the `INCLUDE_DIRS` of those components `REQUIRES` lists (ie all public dependencies of this component's dependencies, recursively expanded).

Main component requirements

The component named `main` is special because it automatically requires all other components in the build. So it's not necessary to pass `REQUIRES` or `PRIV_REQUIRES` to this component. See *renaming main* for a description of what needs to be changed if no longer using the `main` component.

Common component requirements

To avoid duplication, every component automatically requires some "common" IDF components even if they are not mentioned explicitly. Headers from these components can always be included.

The list of common components is: cxx, newlib, freertos, esp_hw_support, heap, log, soc, hal, esp_rom, esp_common, esp_system, xtensa/riscv.

Including components in the build

- By default, every component is included in the build.
- If you set the `COMPONENTS` variable to a minimal list of components used directly by your project, then the build will expand to also include required components. The full list of components will be:
  - Components mentioned explicitly in `COMPONENTS`.
  - Those components' requirements (evaluated recursively).
  - The "common" components that every component depends on.
- Setting `COMPONENTS` to the minimal list of required components can significantly reduce compile times.

Circular Dependencies

It's possible for a project to contain Component A that requires (`REQUIRES` or `PRIV_REQUIRES`) Component B, and Component B that requires Component A. This is known as a dependency cycle or a circular dependency.

CMake will usually handle circular dependencies automatically by repeating the component library names twice on the linker command line. However this strategy doesn't always work, and it's possible the build will fail with a linker error about "Undefined reference to ⋯", referencing a symbol defined by one of the components inside the circular dependency. This is particularly likely if there is a large circular dependency, i.e. A->B->C->D->A.

The best solution is to restructure the components to remove the circular dependency. In most cases, a software architecture without circular dependencies has desirable properties of modularity and clean layering and will be more maintainable in the long term. However, removing circular dependencies is not always possible.

To bypass a linker error caused by a circular dependency, the simplest workaround is to increase the CMake LINK_INTERFACE_MULTIPLICITY property of one of the component libraries. This causes CMake to repeat this library and its dependencies more than two times on the linker command line.

For example:

```
set_property(TARGET ${COMPONENT_LIB} APPEND PROPERTY LINK_INTERFACE_MULTIPLICITY 3)
```

- This line should be placed after `idf_component_register` in the component CMakeLists.txt file.
- If possible, place this line in the component that creates the circular dependency by depending on a lot of other components. However, the line can be placed inside any component that is part of the cycle. Choosing the component that owns the source file shown in the linker error message, or the component that defines the symbol(s) mentioned in the linker error message, is a good place to start.
- Usually increasing the value to 3 (default is 2) is enough, but if this doesn't work then try increasing the number further.
- Adding this option will make the linker command line longer, and the linking stage slower.

Advanced Workaround: Undefined Symbols If only one or two symbols is causing a circular dependency, and all other dependencies are linear, then there is an alternative method to avoid linker errors: Specify the specific symbols required for the "reverse" dependency as undefined symbols at link time.

For example, if component A depends on component B but component B also needs to reference `reverse_ops` from component A (but nothing else), then you can add a line like the following to the component B CMakeLists.txt to resolve the cycle at link time:

```
# This symbol is provided by 'Component A' at link time
target_link_libraries(${COMPONENT_LIB} INTERFACE "-u reverse_ops")
```

- The `-u` argument means that the linker will always include this symbol in the link, regardless of dependency ordering.
- This line should be placed after `idf_component_register` in the component CMakeLists.txt file.
- If 'Component B' doesn't need to access any headers of 'Component A', only link to a few symbol(s), then this line can be used instead of any `REQUIRES` from B to A. This further simplifies the component structure in the build system.

See the target_link_libraries documentation for more information about this CMake function.

Requirements in the build system implementation

- Very early in the CMake configuration process, the script `expand_requirements.cmake` is run. This script does a partial evaluation of all component CMakeLists.txt files and builds a graph of component requirements (this *graph may have cycles*). The graph is used to generate a file `component_depends.cmake` in the build directory.
- The main CMake process then includes this file and uses it to determine the list of components to include in the build (internal `BUILD_COMPONENTS` variable). The `BUILD_COMPONENTS` variable is sorted so dependencies are listed first, however as the component dependency graph has cycles this cannot be guaranteed for all components. The order should be deterministic given the same set of components and component dependencies.
- The value of `BUILD_COMPONENTS` is logged by CMake as "Component names: "
- Configuration is then evaluated for the components included in the build.
- Each component is included in the build normally and the CMakeLists.txt file is evaluated again to add the component libraries to the build.

Component Dependency Order The order of components in the `BUILD_COMPONENTS` variable determines other orderings during the build:

- Order that *project_include.cmake* files are included into the project.
- Order that the list of header paths is generated for compilation (via `-I` argument). (Note that for a given component's source files, only that component's dependency's header paths are passed to the compiler.)

Adding Link-Time Dependencies The ESP-IDF CMake helper function `idf_component_add_link_dependency` adds a link-only dependency between one component and another. In almost all cases, it is better to use the `PRIV_REQUIRES` feature in `idf_component_register` to create a dependency. However, in some cases, it's necessary to add the link-time dependency of another component to this component, i.e., the reverse order to `PRIV_REQUIRES` (for example: *Overriding Default Chip Drivers*).

To make another component depend on this component at link time:

```
idf_component_add_link_dependency(FROM other_component)
```

Place this line after the line with `idf_component_register`.

It's also possible to specify both components by name:

```
idf_component_add_link_dependency(FROM other_component TO that_component)
```

4.5.9 Overriding Parts of the Project

project_include.cmake

For components that have build requirements which must be evaluated before any component CMakeLists files are evaluated, you can create a file called `project_include.cmake` in the component directory. This CMake file is included when `project.cmake` is evaluating the entire project.

`project_include.cmake` files are used inside ESP-IDF, for defining project-wide build features such as `esptool.py` command line arguments and the `bootloader` "special app".

Unlike component `CMakeLists.txt` files, when including a `project_include.cmake` file the current source directory (`CMAKE_CURRENT_SOURCE_DIR` and working directory) is the project directory. Use the variable `COMPONENT_DIR` for the absolute directory of the component.

Note that `project_include.cmake` isn't necessary for the most common component uses - such as adding include directories to the project, or `LDFLAGS` to the final linking step. These values can be customised via the `CMakeLists.txt` file itself. See *Optional Project Variables* for details.

`project_include.cmake` files are included in the order given in `BUILD_COMPONENTS` variable (as logged by CMake). This means that a component's `project_include.cmake` file will be included after it's all dependencies' `project_include.cmake` files, unless both components are part of a dependency cycle. This is important if a `project_include.cmake` file relies on variables set by another component. See also *above*.

Take great care when setting variables or targets in a `project_include.cmake` file. As the values are included into the top-level project CMake pass, they can influence or break functionality across all components!

KConfig.projbuild

This is an equivalent to `project_include.cmake` for *Component Configuration* KConfig files. If you want to include configuration options at the top-level of menuconfig, rather than inside the "Component Configuration" sub-menu, then these can be defined in the KConfig.projbuild file alongside the `CMakeLists.txt` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it's generally better to create a KConfig file for *Component Configuration*.

`project_include.cmake` files are used inside ESP-IDF, for defining project-wide build features such as `esptool.py` command line arguments and the `bootloader` "special app".

4.5.10 Configuration-Only Components

Special components which contain no source files, only `Kconfig.projbuild` and `KConfig`, can have a one-line `CMakeLists.txt` file which calls the function `idf_component_register()` with no arguments specified. This function will include the component in the project build, but no library will be built *and* no header files will be added to any include paths.

4.5.11 Debugging CMake

For full details about CMake and CMake commands, see the CMake v3.16 documentation.

Some tips for debugging the ESP-IDF CMake-based build system:

- When CMake runs, it prints quite a lot of diagnostic information including lists of components and component paths.
- Running `cmake -DDEBUG=1` will produce more verbose diagnostic output from the IDF build system.
- Running `cmake` with the `--trace` or `--trace-expand` options will give a lot of information about control flow. See the cmake command line documentation.

When included from a project CMakeLists file, the `project.cmake` file defines some utility modules and global variables and then sets `IDF_PATH` if it was not set in the system environment.

It also defines an overridden custom version of the built-in CMake `project` function. This function is overridden to add all of the ESP-IDF specific project functionality.

Warning On Undefined Variables

By default, `idf.py` passes the `--warn-uninitialized` flag to CMake so it will print a warning if an undefined variable is referenced in the build. This can be very useful to find buggy CMake files.

If you don't want this behaviour, it can be disabled by passing `--no-warnings` to `idf.py`.

Browse the /tools/cmake/project.cmake file and supporting functions in /tools/cmake/ for more details.

4.5.12 Example Component CMakeLists

Because the build environment tries to set reasonable defaults that will work most of the time, component `CMakeLists.txt` can be very small or even empty (see *Minimal Component CMakeLists*). However, overriding *preset_component_variables* is usually required for some functionality.

Here are some more advanced examples of component CMakeLists files.

Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in the project configuration.

`Kconfig`:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

`CMakeLists.txt`:

```
 set(srcs "foo.c" "more_foo.c")

 if(CONFIG_FOO_ENABLE_BAR)
     list(APPEND srcs "bar.c")
 endif()

idf_component_register(SRCS "${srcs}"
                       ...)
```

This example makes use of the CMake if function and list APPEND function.

This can also be used to select or stub out an implementation, as such:

`Kconfig`:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd

config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

`CMakeLists.txt`:

```
if(CONFIG_ENABLE_LCD_OUTPUT)
   set(srcs lcd-real.c lcd-spi.c)
else()
```

(continues on next page)

(continued from previous page)

```
    set(srcs lcd-dummy.c)
endif()

# We need font if either console or plot is enabled
if(CONFIG_ENABLE_LCD_CONSOLE OR CONFIG_ENABLE_LCD_PLOT)
    list(APPEND srcs "font.c")
endif()

idf_component_register(SRCS "${srcs}"
                       ...)
```

Conditions which depend on the target

The current target is available to CMake files via `IDF_TARGET` variable.

In addition to that, if target `xyz` is used (`IDF_TARGET=xyz`), then Kconfig variable `CONFIG_IDF_TARGET_XYZ` will be set.

Note that component dependencies may depend on `IDF_TARGET` variable, but not on Kconfig variables. Also one can not use Kconfig variables in `include` statements in CMake files, but `IDF_TARGET` can be used in such context.

Source Code Generation

Some components will have a situation where a source file isn't supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called bmp2h. The header file is then included in as C source file called graphics_lib.c:

```
add_custom_command(OUTPUT logo.h
     COMMAND bmp2h -i ${COMPONENT_DIR}/logo.bmp -o log.h
     DEPENDS ${COMPONENT_DIR}/logo.bmp
     VERBATIM)

add_custom_target(logo DEPENDS logo.h)
add_dependencies(${COMPONENT_LIB} logo)

set_property(DIRECTORY "${COMPONENT_DIR}" APPEND PROPERTY
     ADDITIONAL_CLEAN_FILES logo.h)
```

This answer is adapted from the CMake FAQ entry, which contains some other examples that will also work with ESP-IDF builds.

In this example, logo.h will be generated in the current directory (the build directory) while logo.bmp comes with the component and resides under the component path. Because logo.h is a generated file, it should be cleaned when the project is cleaned. For this reason it is added to the ADDITIONAL_CLEAN_FILES property.

Note: If generating files as part of the project CMakeLists.txt file, not a component CMakeLists.txt, then use build property `PROJECT_DIR` instead of `${COMPONENT_DIR}` and `${PROJECT_NAME}.elf` instead of `${COMPONENT_LIB}`.)

If a a source file from another component included `logo.h`, then `add_dependencies` would need to be called to add a dependency between the two components, to ensure that the component source files were always compiled in the correct order.

Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can specify argument `EMBED_FILES` in the component registration, giving space-delimited names of the files to embed:

```
idf_component_register(...
                       EMBED_FILES server_root_cert.der)
```

Or if the file is a string, you can use the variable `EMBED_TXTFILES`. This will embed the contents of the text file as a null-terminated string:

```
idf_component_register(...
                       EMBED_TXTFILES server_root_cert.pem)
```

The file's contents will be added to the .rodata section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_
↪pem_start");
extern const uint8_t server_root_cert_pem_end[]   asm("_binary_server_root_cert_
↪pem_end");
```

The names are generated from the full name of the file, as given in `EMBED_FILES`. Characters /, ., etc. are replaced with underscores. The _binary prefix in the symbol name is added by objcopy and is the same for both text and binary files.

To embed a file into a project, rather than a component, you can call the function `target_add_binary_data` like this:

```
target_add_binary_data(myproject.elf "main/data.bin" TEXT)
```

Place this line after the `project()` line in your project CMakeLists.txt file. Replace `myproject.elf` with your project name. The final argument can be `TEXT` to embed a null-terminated string, or `BINARY` to embed the content as-is.

For an example of using this technique, see the "main" component of the file_serving example protocols/http_server/file_serving/main/CMakeLists.txt - two files are loaded at build time and linked into the firmware.

It is also possible embed a generated file:

```
add_custom_command(OUTPUT my_processed_file.bin
                   COMMAND my_process_file_cmd my_unprocessed_file.bin)
target_add_binary_data(my_target "my_processed_file.bin" BINARY)
```

In the example above, `my_processed_file.bin` is generated from `my_unprocessed_file.bin` through some command `my_process_file_cmd`, then embedded into the target.

To specify a dependence on a target, use the `DEPENDS` argument:

```
add_custom_target(my_process COMMAND ...)
target_add_binary_data(my_target "my_embed_file.bin" BINARY DEPENDS my_process)
```

The `DEPENDS` argument to `target_add_binary_data` ensures that the target executes first.

Code and Data Placements

ESP-IDF has a feature called linker script generation that enables components to define where its code and data will be placed in memory through linker fragment files. These files are processed by the build system, and is used to augment the linker script used for linking app binary. See *Linker Script Generation* for a quick start guide as well as a detailed discussion of the mechanism.

Fully Overriding The Component Build Process

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the ESP-IDF build system entirely by using a CMake feature called ExternalProject. Example component CMakeLists:

```
# External build process for quirc, runs in source dir and
# produces libquirc.a
externalproject_add(quirc_build
    PREFIX ${COMPONENT_DIR}
    SOURCE_DIR ${COMPONENT_DIR}/quirc
    CONFIGURE_COMMAND ""
    BUILD_IN_SOURCE 1
    BUILD_COMMAND make CC=${CMAKE_C_COMPILER} libquirc.a
    INSTALL_COMMAND ""
    )

 # Add libquirc.a to the build process
 add_library(quirc STATIC IMPORTED GLOBAL)
 add_dependencies(quirc quirc_build)

 set_target_properties(quirc PROPERTIES IMPORTED_LOCATION
      ${COMPONENT_DIR}/quirc/libquirc.a)
 set_target_properties(quirc PROPERTIES INTERFACE_INCLUDE_DIRECTORIES
      ${COMPONENT_DIR}/quirc/lib)

 set_directory_properties( PROPERTIES ADDITIONAL_CLEAN_FILES
      "${COMPONENT_DIR}/quirc/libquirc.a")
```

(The above CMakeLists.txt can be used to create a component named `quirc` that builds the quirc project using its own Makefile.)

- `externalproject_add` defines an external build system.
  - `SOURCE_DIR`, `CONFIGURE_COMMAND`, `BUILD_COMMAND` and `INSTALL_COMMAND` should always be set. `CONFIGURE_COMMAND` can be set to an empty string if the build system has no "configure" step. `INSTALL_COMMAND` will generally be empty for ESP-IDF builds.
  - Setting `BUILD_IN_SOURCE` means the build directory is the same as the source directory. Otherwise you can set `BUILD_DIR`.
  - Consult the ExternalProject documentation for more details about `externalproject_add()`
- The second set of commands adds a library target, which points to the "imported" library file built by the external system. Some properties need to be set in order to add include directories and tell CMake where this file is.
- Finally, the generated library is added to ADDITIONAL_CLEAN_FILES. This means `make clean` will delete this library. (Note that the other object files from the build won't be deleted.)

Note: When using an external build process with PSRAM, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See *CONFIG_SPIRAM_CACHE_WORKAROUND* for details of this flag.

ExternalProject dependencies, clean builds CMake has some unusual behaviour around external project builds:

- ADDITIONAL_CLEAN_FILES only works when "make" or "ninja" is used as the build system. If an IDE build system is used, it won't delete these files when cleaning.
- However, the ExternalProject configure & build commands will *always* be re-run after a clean is run.
- Therefore, there are two alternative recommended ways to configure the external build command:
  1. Have the external `BUILD_COMMAND` run a full clean compile of all sources. The build command will be run if any of the dependencies passed to `externalproject_add` with `DEPENDS` have changed, or if this is a clean build (ie any of `idf.py clean`, `ninja clean`, or `make clean` was run.)

2. Have the external `BUILD_COMMAND` be an incremental build command. Pass the parameter `BUILD_ALWAYS 1` to `externalproject_add`. This means the external project will be built each time a build is run, regardless of dependencies. This is only recommended if the external project has correct incremental build behaviour, and doesn't take too long to run.

The best of these approaches for building an external project will depend on the project itself, its build system, and whether you anticipate needing to frequently recompile the project.

4.5.13 Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full sdkconfig configuration, but you do want to override some key values from the ESP-IDF defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when creating a new config from scratch, or when any new config value hasn't yet been set in the `sdkconfig` file.

To override the name of this file or to specify multiple files, set the `SDKCONFIG_DEFAULTS` environment variable or set `SDKCONFIG_DEFAULTS` in top-level `CMakeLists.txt`. File names that are not specified as full paths are resolved relative to current project's diretory.

When specifying multiple files, use a semicolon as the list separator. Files listed first will be applied first. If a particular key is defined in multiple files, the definition in the latter file will overide definitions from former files.

Some of the IDF examples include a `sdkconfig.ci` file. This is part of the continuous integration (CI) test framework and is ignored by the normal build process.

Target-dependent sdkconfig defaults

In addition to `sdkconfig.defaults` file, build system will also load defaults from `sdkconfig.defaults.TARGET_NAME` file, where `TARGET_NAME` is the value of `IDF_TARGET`. For example, for `esp32` target, default settings will be taken from `sdkconfig.defaults` first, and then from `sdkconfig.defaults.esp32`.

If `SDKCONFIG_DEFAULTS` is used to override the name of defaults file/files, the name of target-specific defaults file will be derived from `SDKCONFIG_DEFAULTS` value/values using the rule above. When there are multiple files in `SDKCONFIG_DEFAULTS`, target-specific file will be applied right after the file bringing it in, before all latter files in `SDKCONFIG_DEFAULTS`

For example, if `SDKCONFIG_DEFAULTS="sdkconfig.defaults;sdkconfig_devkit1"`, and there is a file `sdkconfig.defaults.esp32` in the same folder, then the files will be applied in the following order: (1) sdkconfig.defaults (2) sdkconfig.defaults.esp32 (3) sdkconfig_devkit1.

4.5.14 Flash arguments

There are some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, esptool.py and esptool write_flash arguments. It's simple to write a script to save binaries and esptool.py.

After running a project build, the build directory contains binary output files (`.bin` files) for the project and also the following flashing data files:

- `flash_project_args` contains arguments to flash the entire project (app, bootloader, partition table, PHY data if this is configured).
- `flash_app_args` contains arguments to flash only the app.
- `flash_bootloader_args` contains arguments to flash only the bootloader.

You can pass any of these flasher argument files to `esptool.py` as follows:

```
python esptool.py --chip esp32 write_flash @build/flash_project_args
```

Alternatively, it is possible to manually copy the parameters from the argument file and pass them on the command line.

The build directory also contains a generated file `flasher_args.json` which contains project flash information, in JSON format. This file is used by `idf.py` and can also be used by other tools which need information about the project build.

4.5.15 Building the Bootloader

The bootloader is a special "subproject" inside /components/bootloader/subproject. It has its own project CMakeLists.txt file and builds separate .ELF and .BIN files to the main project. However it shares its configuration and build directory with the main project.

The subproject is inserted as an external project from the top-level project, by the file /components/bootloader/project_include.cmake. The main build process runs CMake for the subproject, which includes discovering components (a subset of the main components) and generating a bootloader-specific config (derived from the main `sdkconfig`).

4.5.16 Writing Pure CMake Components

The ESP-IDF build system "wraps" CMake with the concept of "components", and helper functions to automatically integrate these components into a project build.

However, underneath the concept of "components" is a full CMake build system. It is also possible to make a component which is pure CMake.

Here is an example minimal "pure CMake" component CMakeLists file for a component named `json`:

```
add_library(json STATIC
cJSON/cJSON.c
cJSON/cJSON_Utils.c)

target_include_directories(json PUBLIC cJSON)
```

- This is actually an equivalent declaration to the IDF `json` component /components/json/CMakeLists.txt.
- This file is quite simple as there are not a lot of source files. For components with a large number of files, the globbing behaviour of ESP-IDF's component logic can make the component CMakeLists style simpler.)
- Any time a component adds a library target with the component name, the ESP-IDF build system will automatically add this to the build, expose public include directories, etc. If a component wants to add a library target with a different name, dependencies will need to be added manually via CMake commands.

4.5.17 Using Third-Party CMake Projects with Components

CMake is used for a lot of open-source C and C++ projects —code that users can tap into for their applications. One of the benefits of having a CMake build system is the ability to import these third-party projects, sometimes even without modification! This allows for users to be able to get functionality that may not yet be provided by a component, or use another library for the same functionality.

Importing a library might look like this for a hypothetical library `foo` to be used in the `main` component:

```
# Register the component
idf_component_register(...)

# Set values of hypothetical variables that control the build of `foo`
set(FOO_BUILD_STATIC OFF)
set(FOO_BUILD_TESTS OFF)

# Create and import the library targets
add_subdirectory(foo)

# Publicly link `foo` to `main` component
target_link_libraries(main PUBLIC foo)
```

For an actual example, take a look at build_system/cmake/import_lib. Take note that what needs to be done in order to import the library may vary. It is recommended to read up on the library's documentation for instructions on how to import it from other projects. Studying the library's CMakeLists.txt and build structure can also be helpful.

It is also possible to wrap a third-party library to be used as a component in this manner. For example, the mbedtls component is a wrapper for Espressif's fork of mbedtls. See its component CMakeLists.txt .

The CMake variable `ESP_PLATFORM` is set to 1 whenever the ESP-IDF build system is being used. Tests such as `if (ESP_PLATFORM)` can be used in generic CMake code if special IDF-specific logic is required.

Using ESP-IDF components from external libraries

The above example assumes that the external library `foo` (or `tinyxml` in the case of the `import_lib` example) doesn't need to use any ESP-IDF APIs apart from common APIs such as libc, libstdc++, etc. If the external library needs to use APIs provided by other ESP-IDF components, this needs to be specified in the external CMakeLists.txt file by adding a dependency on the library target `idf::<componentname>`.

For example, in the `foo/CMakeLists.txt` file:

```
add_library(foo bar.c fizz.cpp buzz.cpp)

if(ESP_PLATFORM)
  # On ESP-IDF, bar.c needs to include esp_flash.h from the spi_flash component
  target_link_libraries(foo PRIVATE idf::spi_flash)
endif()
```

4.5.18 Using Prebuilt Libraries with Components

Another possibility is that you have a prebuilt static library (`.a` file), built by some other build process.

The ESP-IDF build system provides a utility function `add_prebuilt_library` for users to be able to easily import and use prebuilt libraries:

```
add_prebuilt_library(target_name lib_path [REQUIRES req1 req2 ...] [PRIV_REQUIRES␣
↪req1 req2 ...])
```

where:

- `target_name`- name that can be used to reference the imported library, such as when linking to other targets
- `lib_path`- path to prebuilt library; may be an absolute or relative path to the component directory

Optional arguments `REQUIRES` and `PRIV_REQUIRES` specify dependency on other components. These have the same meaning as the arguments for `idf_component_register`.

Take note that the prebuilt library must have been compiled for the same target as the consuming project. Configuration relevant to the prebuilt library must also match. If not paid attention to, these two factors may contribute to subtle bugs in the app.

For an example, take a look at build_system/cmake/import_prebuilt.

4.5.19 Using ESP-IDF in Custom CMake Projects

ESP-IDF provides a template CMake project for easily creating an application. However, in some instances the user might already have an existing CMake project or may want to create a custom one. In these cases it is desirable to be able to consume IDF components as libraries to be linked to the user's targets (libraries/ executables).

It is possible to do so by using the *build system APIs provided* by tools/cmake/idf.cmake. For example:

```
cmake_minimum_required(VERSION 3.16)
project(my_custom_app C)

# Include CMake file that provides ESP-IDF CMake build system APIs.
include($ENV{IDF_PATH}/tools/cmake/idf.cmake)

# Include ESP-IDF components in the build, may be thought as an equivalent of
# add_subdirectory() but with some additional processing and magic for ESP-IDF␣
↪build
# specific build processes.
idf_build_process(esp32)

# Create the project executable and plainly link the newlib component to it using
# its alias, idf::newlib.
add_executable(${CMAKE_PROJECT_NAME}.elf main.c)
target_link_libraries(${CMAKE_PROJECT_NAME}.elf idf::newlib)

# Let the build system know what the project executable is to attach more targets,␣
↪dependencies, etc.
idf_build_executable(${CMAKE_PROJECT_NAME}.elf)
```

The example in build_system/cmake/idf_as_lib demonstrates the creation of an application equivalent to hello world application using a custom CMake project.

Note: The IDF build system can only set compiler flags for source files that it builds. When an external CMakeLists.txt file is used and PSRAM is enabled, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See *CONFIG_SPIRAM_CACHE_WORKAROUND* for details of this flag.

4.5.20 ESP-IDF CMake Build System API

idf-build-commands

```
idf_build_get_property(var property [GENERATOR_EXPRESSION])
```

Retrieve a *build property* *property* and store it in *var* accessible from the current scope. Specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that property, instead of the actual value, which can be used with CMake commands that support generator expressions.

```
idf_build_set_property(property val [APPEND])
```

Set a *build property* *property* with value *val*. Specifying *APPEND* will append the specified value to the current value of the property. If the property does not previously exist or it is currently empty, the specified value becomes the first element/member instead.

```
idf_build_component(component_dir)
```

Present a directory *component_dir* that contains a component to the build system. Relative paths are converted to absolute paths with respect to current directory. All calls to this command must be performed before *idf_build_process*.

This command does not guarantee that the component will be processed during build (see the *COMPONENTS* argument description for *idf_build_process*)

```
idf_build_process(target
                  [PROJECT_DIR project_dir]
                  [PROJECT_VER project_ver]
                  [PROJECT_NAME project_name]
                  [SDKCONFIG sdkconfig]
```

(continues on next page)

(continued from previous page)

```
                    [SDKCONFIG_DEFAULTS sdkconfig_defaults]
                    [BUILD_DIR build_dir]
                    [COMPONENTS component1 component2 ...])
```

Performs the bulk of the behind-the-scenes magic for including ESP-IDF components such as component configuration, libraries creation, dependency expansion and resolution. Among these functions, perhaps the most important from a user's perspective is the libraries creation by calling each component's `idf_component_register`. This command creates the libraries for each component, which are accessible using aliases in the form idf::*component_name*. These aliases can be used to link the components to the user's own targets, either libraries or executables.

The call requires the target chip to be specified with *target* argument. Optional arguments for the call include:

- PROJECT_DIR - directory of the project; defaults to CMAKE_SOURCE_DIR
- PROJECT_NAME - name of the project; defaults to CMAKE_PROJECT_NAME
- PROJECT_VER - version/revision of the project; defaults to "1"
- SDKCONFIG - output path of generated sdkconfig file; defaults to PROJECT_DIR/sdkconfig or CMAKE_SOURCE_DIR/sdkconfig depending if PROJECT_DIR is set
- SDKCONFIG_DEFAULTS - list of files containing default config to use in the build (list must contain full paths); defaults to empty. For each value *filename* in the list, the config from file *filename.target*, if it exists, is also loaded.
- BUILD_DIR - directory to place ESP-IDF build-related artifacts, such as generated binaries, text files, components; defaults to CMAKE_BINARY_DIR
- COMPONENTS - select components to process among the components known by the build system (added via *idf_build_component*). This argument is used to trim the build. Other components are automatically added if they are required in the dependency chain, i.e. the public and private requirements of the components in this list are automatically added, and in turn the public and private requirements of those requirements, so on and so forth. If not specified, all components known to the build system are processed.

```
idf_build_executable(executable)
```

Specify the executable *executable* for ESP-IDF build. This attaches additional targets such as dependencies related to flashing, generating additional binary files, etc. Should be called after `idf_build_process`.

```
idf_build_get_config(var config [GENERATOR_EXPRESSION])
```

Get the value of the specified config. Much like build properties, specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that config, instead of the actual value, which can be used with CMake commands that support generator expressions. Actual config values are only known after call to `idf_build_process`, however.

idf-build-properties

These are properties that describe the build. Values of build properties can be retrieved by using the build command `idf_build_get_property`. For example, to get the Python interpreter used for the build:

```
idf_build_get_property(python PYTHON)
message(STATUS "The Python intepreter is: ${python}")
```

- BUILD_DIR - build directory; set from `idf_build_process` BUILD_DIR argument
- BUILD_COMPONENTS - list of components included in the build; set by `idf_build_process`
- BUILD_COMPONENT_ALIASES - list of library alias of components included in the build; set by `idf_build_process`
- C_COMPILE_OPTIONS - compile options applied to all components' C source files
- COMPILE_OPTIONS - compile options applied to all components' source files, regardless of it being C or C++
- COMPILE_DEFINITIONS - compile definitions applied to all component source files
- CXX_COMPILE_OPTIONS - compile options applied to all components' C++ source files

- EXECUTABLE - project executable; set by call to `idf_build_executable`
- EXECUTABLE_NAME - name of project executable without extension; set by call to `idf_build_executable`
- EXECUTABLE_DIR - path containing the output executable
- IDF_COMPONENT_MANAGER - the component manager is enabled by default, but if this property is set to `0` it was disabled by the IDF_COMPONENT_MANAGER environment variable
- IDF_PATH - ESP-IDF path; set from IDF_PATH environment variable, if not, inferred from the location of `idf.cmake`
- IDF_TARGET - target chip for the build; set from the required target argument for `idf_build_process`
- IDF_VER - ESP-IDF version; set from either a version file or the Git revision of the IDF_PATH repository
- INCLUDE_DIRECTORIES - include directories for all component source files
- KCONFIGS - list of Kconfig files found in components in build; set by `idf_build_process`
- KCONFIG_PROJBUILDS - list of Kconfig.projbuild files found in components in build; set by `idf_build_process`
- PROJECT_NAME - name of the project; set from `idf_build_process` PROJECT_NAME argument
- PROJECT_DIR - directory of the project; set from `idf_build_process` PROJECT_DIR argument
- PROJECT_VER - version of the project; set from `idf_build_process` PROJECT_VER argument
- PYTHON - Python interpreter used for the build; set from PYTHON environment variable if available, if not "python" is used
- SDKCONFIG - full path to output config file; set from `idf_build_process` SDKCONFIG argument
- SDKCONFIG_DEFAULTS - list of files containing default config to use in the build; set from `idf_build_process` SDKCONFIG_DEFAULTS argument
- SDKCONFIG_HEADER - full path to C/C++ header file containing component configuration; set by `idf_build_process`
- SDKCONFIG_CMAKE - full path to CMake file containing component configuration; set by `idf_build_process`
- SDKCONFIG_JSON - full path to JSON file containing component configuration; set by `idf_build_process`
- SDKCONFIG_JSON_MENUS - full path to JSON file containing config menus; set by `idf_build_process`

idf-component-commands

```
idf_component_get_property(var component property [GENERATOR_EXPRESSION])
```

Retrieve a specified *component*'s *component property*, *property* and store it in *var* accessible from the current scope. Specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that property, instead of the actual value, which can be used with CMake commands that support generator expressions.

```
idf_component_set_property(component property val [APPEND])
```

Set a specified *component*'s *component property*, *property* with value *val*. Specifying *APPEND* will append the specified value to the current value of the property. If the property does not previously exist or it is currently empty, the specified value becomes the first element/member instead.

```
idf_component_register([[SRCS src1 src2 ...] | [[SRC_DIRS dir1 dir2 ...] [EXCLUDE_
↪SRCS src1 src2 ...]]
                       [INCLUDE_DIRS dir1 dir2 ...]
                       [PRIV_INCLUDE_DIRS dir1 dir2 ...]
                       [REQUIRES component1 component2 ...]
                       [PRIV_REQUIRES component1 component2 ...]
                       [LDFRAGMENTS ldfragment1 ldfragment2 ...]
                       [REQUIRED_IDF_TARGETS target1 target2 ...]
                       [EMBED_FILES file1 file2 ...]
                       [EMBED_TXTFILES file1 file2 ...]
                       [KCONFIG kconfig]
                       [KCONFIG_PROJBUILD kconfig_projbuild]
                       [WHOLE_ARCHIVE])
```

Register a component to the build system. Much like the `project()` CMake command, this should be called from the component's CMakeLists.txt directly (not through a function or macro) and is recommended to be called before any other command. Here are some guidelines on what commands can **not** be called before `idf_component_register`:

- commands that are not valid in CMake script mode
- custom commands defined in project_include.cmake
- build system API commands except `idf_build_get_property`; although consider whether the property may not have been set yet

Commands that set and operate on variables are generally okay to call before `idf_component_register`.

The arguments for `idf_component_register` include:

- SRCS - component source files used for creating a static library for the component; if not specified, component is a treated as a config-only component and an interface library is created instead.
- SRC_DIRS, EXCLUDE_SRCS - used to glob source files (.c, .cpp, .S) by specifying directories, instead of specifying source files manually via SRCS. Note that this is subject to the *limitations of globbing in CMake*. Source files specified in EXCLUDE_SRCS are removed from the globbed files.
- INCLUDE_DIRS - paths, relative to the component directory, which will be added to the include search path for all other components which require the current component
- PRIV_INCLUDE_DIRS - directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only
- REQUIRES - public component requirements for the component
- PRIV_REQUIRES - private component requirements for the component; ignored on config-only components
- LDFRAGMENTS - component linker fragment files
- REQUIRED_IDF_TARGETS - specify the only target the component supports
- KCONFIG - override the default Kconfig file
- KCONFIG_PROJBUILD - override the default Kconfig.projbuild file
- WHOLE_ARCHIVE - if specified, the component library is surrounded by `-Wl,--whole-archive`, `-Wl,--no-whole-archive` when linked. This has the same effect as setting `WHOLE_ARCHIVE` component property.

The following are used for *embedding data into the component*, and is considered as source files when determining if a component is config-only. This means that even if the component does not specify source files, a static library is still created internally for the component if it specifies either:

- EMBED_FILES - binary files to be embedded in the component
- EMBED_TXTFILES - text files to be embedded in the component

idf-component-properties

These are properties that describe a component. Values of component properties can be retrieved by using the build command `idf_component_get_property`. For example, to get the directory of the `freertos` component:

```
idf_component_get_property(dir freertos COMPONENT_DIR)
message(STATUS "The 'freertos' component directory is: ${dir}")
```

- COMPONENT_ALIAS - alias for COMPONENT_LIB used for linking the component to external targets; set by `idf_build_component` and alias library itself is created by `idf_component_register`
- COMPONENT_DIR - component directory; set by `idf_build_component`
- COMPONENT_OVERRIDEN_DIR - contains the directory of the original component if *this component overrides another component*
- COMPONENT_LIB - name for created component static/interface library; set by `idf_build_component` and library itself is created by `idf_component_register`
- COMPONENT_NAME - name of the component; set by `idf_build_component` based on the component directory name
- COMPONENT_TYPE - type of the component, whether LIBRARY or CONFIG_ONLY. A component is of type LIBRARY if it specifies source files or embeds a file
- EMBED_FILES - list of files to embed in component; set from `idf_component_register` EMBED_FILES argument

- EMBED_TXTFILES - list of text files to embed in component; set from `idf_component_register` EMBED_TXTFILES argument
- INCLUDE_DIRS - list of component include directories; set from `idf_component_register` INCLUDE_DIRS argument
- KCONFIG - component Kconfig file; set by `idf_build_component`
- KCONFIG_PROJBUILD - component Kconfig.projbuild; set by `idf_build_component`
- LDFRAGMENTS - list of component linker fragment files; set from `idf_component_register` LDFRAGMENTS argument
- MANAGED_PRIV_REQUIRES - list of private component dependencies added by the IDF component manager from dependencies in `idf_component.yml` manifest file
- MANAGED_REQUIRES - list of public component dependencies added by the IDF component manager from dependencies in `idf_component.yml` manifest file
- PRIV_INCLUDE_DIRS - list of component private include directories; set from `idf_component_register` PRIV_INCLUDE_DIRS on components of type LIBRARY
- PRIV_REQUIRES - list of private component dependentices; set from value of `idf_component_register` PRIV_REQUIRES argument and dependencies in `idf_component.yml` manifest file
- REQUIRED_IDF_TARGETS - list of targets the component supports; set from `idf_component_register` EMBED_TXTFILES argument
- REQUIRES - list of public component dependencies; set from value of `idf_component_register` REQUIRES argument and dependencies in `idf_component.yml` manifest file
- SRCS - list of component source files; set from SRCS or SRC_DIRS/EXCLUDE_SRCS argument of `idf_component_register`
- WHOLE_ARCHIVE - if this property is set to `TRUE` (or any boolean "true" CMake value: 1, `ON`, `YES`, `Y`), the component library is surrounded by `-Wl,--whole-archive`, `-Wl,--no-whole-archive` when linked. This can be used to force the linker to include every object file into the executable, even if the object file doesn't resolve any references from the rest of the application. This is commonly used when a component contains plugins or modules which rely on link-time registration. This property is `FALSE` by default. It can be set to `TRUE` from the component CMakeLists.txt file.

4.5.21 File Globbing & Incremental Builds

The preferred way to include source files in an ESP-IDF component is to list them manually via SRCS argument to `idf_component_register`:

```
idf_component_register(SRCS library/a.c library/b.c platform/platform.c
                       ...)
```

This preference reflects the CMake best practice of manually listing source files. This could, however, be inconvenient when there are lots of source files to add to the build. The ESP-IDF build system provides an alternative way for specifying source files using `SRC_DIRS`:

```
idf_component_register(SRC_DIRS library platform
                       ...)
```

This uses globbing behind the scenes to find source files in the specified directories. Be aware, however, that if a new source file is added and this method is used, then CMake won't know to automatically re-run and this file won't be added to the build.

The trade-off is acceptable when you're adding the file yourself, because you can trigger a clean build or run `idf.py reconfigure` to manually re-run CMake. However, the problem gets harder when you share your project with others who may check out a new version using a source control tool like Git…

For components which are part of ESP-IDF, we use a third party Git CMake integration module (/tools/cmake/third_party/GetGitRevisionDescription.cmake) which automatically re-runs CMake any time the repository commit changes. This means if you check out a new ESP-IDF version, CMake will automatically re-run.

For project components (not part of ESP-IDF), there are a few different options:

- If keeping your project file in Git, ESP-IDF will automatically track the Git revision and re-run CMake if the revision changes.
- If some components are kept in a third git repository (not the project repository or ESP-IDF repository), you can add a call to the `git_describe` function in a component CMakeLists file in order to automatically trigger re-runs of CMake when the Git revision changes.
- If not using Git, remember to manually run `idf.py reconfigure` whenever a source file may change.
- To avoid this problem entirely, use `SRCS` argument to `idf_component_register` to list all source files in project components.

The best option will depend on your particular project and its users.

4.5.22 Build System Metadata

For integration into IDEs and other build systems, when CMake runs the build process generates a number of metadata files in the `build/` directory. To regenerate these files, run `cmake` or `idf.py reconfigure` (or any other `idf.py` build command).

- `compile_commands.json` is a standard format JSON file which describes every source file which is compiled in the project. A CMake feature generates this file, and many IDEs know how to parse it.
- `project_description.json` contains some general information about the ESP-IDF project, configured paths, etc.
- `flasher_args.json` contains esptool.py arguments to flash the project's binary files. There are also `flash_*_args` files which can be used directly with esptool.py. See *Flash arguments*.
- `CMakeCache.txt` is the CMake cache file which contains other information about the CMake process, toolchain, etc.
- `config/sdkconfig.json` is a JSON-formatted version of the project configuration values.
- `config/kconfig_menus.json` is a JSON-formatted version of the menus shown in menuconfig, for use in external IDE UIs.

JSON Configuration Server

A tool called `confserver.py` is provided to allow IDEs to easily integrate with the configuration system logic. `confserver.py` is designed to run in the background and interact with a calling process by reading and writing JSON over process stdin & stdout.

You can run `confserver.py` from a project via `idf.py confserver` or `ninja confserver`, or a similar target triggered from a different build generator.

For more information about `confserver.py`, see tools/kconfig_new/README.md.

4.5.23 Build System Internals

Build Scripts

The listfiles for the ESP-IDF build system reside in /tools/cmake. The modules which implement core build system functionality are as follows:

- build.cmake - Build related commands i.e. build initialization, retrieving/setting build properties, build processing.
- component.cmake - Component related commands i.e. adding components, retrieving/setting component properties, registering components.
- kconfig.cmake - Generation of configuration files (sdkconfig, sdkconfig.h, sdkconfig.cmake, etc.) from Kconfig files.
- ldgen.cmake - Generation of final linker script from linker fragment files.
- target.cmake - Setting build target and toolchain file.
- utilities.cmake - Miscellaneous helper commands.

Aside from these files, there are two other important CMake scripts in /tools/cmake:

- idf.cmake - Sets up the build and includes the core modules listed above. Included in CMake projects in order to access ESP-IDF build system functionality.
- project.cmake - Includes `idf.cmake` and provides a custom `project()` command that takes care of all the heavy lifting of building an executable. Included in the top-level CMakeLists.txt of standard ESP-IDF projects.

The rest of the files in /tools/cmake are support or third-party scripts used in the build process.

Build Process

This section describes the standard ESP-IDF application build process. The build process can be broken down roughly into four phases:

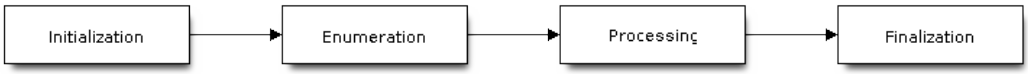


Fig. 3: ESP-IDF Build System Process

Initialization This phase sets up necessary parameters for the build.

- **Upon inclusion of `idf.cmake` in `project.cmake`, the following steps are performed:**
  - Set `IDF_PATH` from environment variable or inferred from path to `project.cmake` included in the top-level CMakeLists.txt.
  - Add /tools/cmake to `CMAKE_MODULE_PATH` and include core modules plus the various helper/third-party scripts.
  - Set build tools/executables such as default Python interpreter.
  - Get ESP-IDF git revision and store as `IDF_VER`.
  - Set global build specifications i.e. compile options, compile definitions, include directories for all components in the build.
  - Add components in components to the build.
- **The initial part of the custom `project()` command performs the following steps:**
  - Set `IDF_TARGET` from environment variable or CMake cache and the corresponding `CMAKE_TOOLCHAIN_FILE` to be used.
  - Add components in `EXTRA_COMPONENT_DIRS` to the build.
  - Prepare arguments for calling command `idf_build_process()` from variables such as `COMPONENTS`/`EXCLUDE_COMPONENTS`, `SDKCONFIG`, `SDKCONFIG_DEFAULTS`.

The call to `idf_build_process()` command marks the end of this phase.

Enumeration

This phase builds a final list of components to be processed in the build, and is performed in the first half of `idf_build_process()`.

- Retrieve each component's public and private requirements. A child process is created which executes each component's CMakeLists.txt in script mode. The values of `idf_component_register` REQUIRES and PRIV_REQUIRES argument is returned to the parent build process. This is called early expansion. The variable `CMAKE_BUILD_EARLY_EXPANSION` is defined during this step.
- Recursively include components based on public and private requirements.

Processing

This phase processes the components in the build, and is the second half of `idf_build_process()`.

- Load project configuration from sdkconfig file and generate an sdkconfig.cmake and sdkconfig.h header. These define configuration variables/macros that are accessible from the build scripts and C/C++ source/header files, respectively.
- Include each component's `project_include.cmake`.
- Add each component as a subdirectory, processing its CMakeLists.txt. The component CMakeLists.txt calls the registration command, `idf_component_register` which adds source files, include directories, creates component library, links dependencies, etc.

Finalization

This phase is everything after `idf_build_process()`.

- Create executable and link the component libraries to it.
- Generate project metadata files such as project_description.json and display relevant information about the project built.

Browse /tools/cmake/project.cmake for more details.

4.5.24 Migrating from ESP-IDF GNU Make System

Some aspects of the CMake-based ESP-IDF build system are very similar to the older GNU Make-based system. The developer needs to provide values the include directories, source files etc. There is a syntactical difference, however, as the developer needs to pass these as arguments to the registration command, `idf_component_register`.

Automatic Conversion Tool

An automatic project conversion tool is available in *tools/cmake/convert_to_cmake.py* in ESP-IDF v4.x releases. The script was removed in v5.0 because of its *make* build system dependency.

No Longer Available in CMake

Some features are significantly different or removed in the CMake-based system. The following variables no longer exist in the CMake-based build system:

- `COMPONENT_BUILD_DIR`: Use `CMAKE_CURRENT_BINARY_DIR` instead.
- `COMPONENT_LIBRARY`: Defaulted to `$(COMPONENT_NAME).a`, but the library name could be overriden by the component. The name of the component library can no longer be overriden by the component.
- `CC`, `LD`, `AR`, `OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain. Use `CMAKE_C_COMPILER`, `CMAKE_C_LINK_EXECUTABLE`, `CMAKE_OBJCOPY`, etc instead. Full list here.
- `HOSTCC`, `HOSTLD`, `HOSTAR`: Full names of each tool from the host native toolchain. These are no longer provided, external projects should detect any required host toolchain manually.
- `COMPONENT_ADD_LDFLAGS`: Used to override linker flags. Use the CMake target_link_libraries command instead.
- `COMPONENT_ADD_LINKER_DEPS`: List of files that linking should depend on. target_link_libraries will usually infer these dependencies automatically. For linker scripts, use the provided custom CMake function `target_linker_scripts`.
- `COMPONENT_SUBMODULES`: No longer used, the build system will automatically enumerate all submodules in the ESP-IDF repository.
- `COMPONENT_EXTRA_INCLUDES`: Used to be an alternative to `COMPONENT_PRIV_INCLUDEDIRS` for absolute paths. Use `PRIV_INCLUDE_DIRS` argument to `idf_component_register` for all cases now (can be relative or absolute).
- `COMPONENT_OBJS`: Previously, component sources could be specified as a list of object files. Now they can be specified as a list of source files via `SRCS` argument to *idf_component_register*.

- `COMPONENT_OBJEXCLUDE`: Has been replaced with `EXCLUDE_SRCS` argument to `idf_component_register`. Specify source files (as absolute paths or relative to component directory), instead.
- `COMPONENT_EXTRA_CLEAN`: Set property `ADDITIONAL_CLEAN_FILES` instead but note *CMake has some restrictions around this functionality*.
- `COMPONENT_OWNBUILDTARGET` & `COMPONENT_OWNCLEANTARGET`: Use CMake ExternalProject instead. See *Fully Overriding The Component Build Process* for full details.
- `COMPONENT_CONFIG_ONLY`: Call `idf_component_register` without any arguments instead. See *Configuration-Only Components*.
- `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`: Use equivalent CMake commands instead. See *Controlling Component Compilation*.

No Default Values

Unlike in the legacy Make-based build system, the following have no default values:

- Source directories (`COMPONENT_SRCDIRS` variable in Make, `SRC_DIRS` argument to `idf_component_register` in CMake)
- Include directories (`COMPONENT_ADD_INCLUDEDIRS` variable in Make, `INCLUDE_DIRS` argument to `idf_component_register` in CMake)

No Longer Necessary

- In the legacy Make-based build system, it is required to also set `COMPONENT_SRCDIRS` if `COMPONENT_SRCS` is set. In CMake, the equivalent is not necessary i.e. specifying `SRC_DIRS` to `idf_component_register` if `SRCS` is also specified (in fact, `SRCS` is ignored if `SRC_DIRS` is specified).

Flashing from make

`make flash` and similar targets still work to build and flash. However, project `sdkconfig` no longer specifies serial port and baud rate. Environment variables can be used to override these. See *Flashing with ninja or make* for more details.

4.6 Core Dump

4.6.1 Overview

ESP-IDF provides support to generate core dumps on unrecoverable software errors. This useful technique allows post-mortem analysis of software state at the moment of failure. Upon the crash system enters panic state, prints some information and halts or reboots depending configuration. User can choose to generate core dump in order to analyse the reason of failure on PC later on. Core dump contains snapshots of all tasks in the system at the moment of failure. Snapshots include tasks control blocks (TCB) and stacks. So it is possible to find out what task, at what instruction (line of code) and what callstack of that task lead to the crash. It is also possible dumping variables content on demand if previously attributed accordingly. ESP-IDF provides special script *espcoredump.py* to help users to retrieve and analyse core dumps. This tool provides two commands for core dumps analysis:

- `info_corefile` - prints crashed task' s registers, callstack, list of available tasks in the system, memory regions and contents of memory stored in core dump (TCBs and stacks)
- `dbg_corefile` - creates core dump ELF file and runs GDB debug session with this file. User can examine memory, variables and tasks states manually. Note that since not all memory is saved in core dump only values of variables allocated on stack will be meaningful

For more information about core dump internals see the - *Core dump internals*

4.6.2 Configurations

There are a number of core dump related configuration options which user can choose in project configuration menu (`idf.py menuconfig`).

Core dump data destination (Components -> Core dump -> Data destination)

- Save core dump to Flash (Flash)
- Print core dump to UART (UART)
- Disable core dump generation (None)

Core dump data format (Components -> Core dump -> Core dump data format)

- ELF format (Executable and Linkable Format file for core dump)
- Binary format (Basic binary format for core dump)

The ELF format contains extended features and allow to save more information about broken tasks and crashed software but it requires more space in the flash memory. This format of core dump is recommended for new software designs and is flexible enough to extend saved information for future revisions.

The Binary format is kept for compatibility reasons, it uses less space in the memory to keep data and provides better performance.

Core dump data integrity check (Components -> Core dump -> Core dump data integrity check)

- Use CRC32 for core dump integrity verification
- Use SHA256 for core dump integrity verification (only work in ELF format)

The CRC32 option provides better calculation performance and consumes less memory for storage.

The SHA256 hash algorithm provides greater probability of detecting corruption than a CRC32 with multiple bit errors.

Maximum number of tasks snapshots in core dump (Components -> Core dump -> Maximum number of tasks)

Delay before core dump is printed to UART (Components -> Core dump -> Delay before print to UART)

The value is in ms.

Handling of UART core dumps in IDF Monitor (Components -> Core dump -> Delay before print to UART)

The value is base64 encoded.

- Decode and show summary (info_corefile)
- Don’ t decode

Reserved stack size (Components -> Core dump -> Reserved stack size)

Size of the memory to be reserved for core dump stack. If 0 core dump process will run on the stack of crashed task/ISR, otherwise special stack will be allocated. To ensure that core dump itself will not overflow task/ISR stack set this to the value above 800.

4.6.3 Save core dump to flash

When this option is selected core dumps are saved to special partition on flash. When using default partition table files which are provided with ESP-IDF it automatically allocates necessary space on flash, But if user wants to use its own layout file together with core dump feature it should define separate partition for core dump as it is shown below:

```
# Name,   Type, SubType, Offset,  Size
# Note: if you have increased the bootloader size, make sure to update the offsets␣
↪to avoid overlap
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
```

(continues on next page)

(continued from previous page)

```
factory,  app,  factory, 0x10000, 1M
coredump, data, coredump,,        64K
```

There are no special requirements for partition name. It can be chosen according to the user application needs, but partition type should be ‘data’ and sub-type should be ‘coredump’. Also when choosing partition size note that core dump data structure introduces constant overhead of 20 bytes and per-task overhead of 12 bytes. This overhead does not include size of TCB and stack for every task. So partition size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

The example of generic command to analyze core dump from flash is:

```
espcoredump.py -p </path/to/serial/port> info_corefile </path/to/program/elf/file>
```

or

```
espcoredump.py -p </path/to/serial/port> dbg_corefile </path/to/program/elf/file>
```

4.6.4 Print core dump to UART

When this option is selected base64-encoded core dumps are printed on UART upon system panic. In this case user should save core dump text body to some file manually and then run the following command:

```
espcoredump.py --chip esp32 info_corefile -t b64 -c </path/to/saved/base64/text> </
↪path/to/program/elf/file>
```

or

```
espcoredump.py --chip esp32 dbg_corefile -t b64 -c </path/to/saved/base64/text> </
↪path/to/program/elf/file>
```

Base64-encoded body of core dump will be between the following header and footer:

```
================ CORE DUMP START =================
<body of base64-encoded core dump, save it to file on disk>
================= CORE DUMP END ==================
```

The `CORE DUMP START` and `CORE DUMP END` lines must not be included in core dump text file.

4.6.5 ROM Functions in Backtraces

It is possible situation that at the moment of crash some tasks or/and crashed task itself have one or more ROM functions in their callstacks. Since ROM is not part of the program ELF it will be impossible for GDB to parse such callstacks, because it tries to analyse functions’ prologues to accomplish that. In that case callstack printing will be broken with error message at the first ROM function. To overcome this issue, you can use the ROM ELF provided by Espressif. You can find the esp32’s corresponding ROM ELF file from the list of released archives. The ROM ELF file can then be passed to `espcoredump.py`. More details about ROM ELFs can be found here.

4.6.6 Dumping variables on demand

Sometimes you want to read the last value of a variable to understand the root cause of a crash. Core dump supports retrieving variable data over GDB by attributing special notations declared variables.

Supported notations and RAM regions

- `COREDUMP_DRAM_ATTR` places variable into DRAM area which will be included into dump.
- `COREDUMP_RTC_ATTR` places variable into RTC area which will be included into dump.
- `COREDUMP_RTC_FAST_ATTR` places variable into RTC_FAST area which will be included into dump.

Example

1. In *Project Configuration Menu*, enable *COREDUMP TO FLASH*, then save and exit.
2. In your project, create a global variable in DRAM area as such as:

```
// uint8_t global_var;
COREDUMP_DRAM_ATTR uint8_t global_var;
```

3. In main application, set the variable to any value and `assert(0)` to cause a crash.

```
global_var = 25;
assert(0);
```

4. Build, flash and run the application on a target device and wait for the dumping information.
5. Run the command below to start core dumping in GDB, where `PORT` is the device USB port:

```
espcoredump.py -p PORT dbg_corefile <path/to/elf>
```

6. In GDB shell, type `p global_var` to get the variable content:

```
(gdb) p global_var
$1 = 25 '\031'
```

4.6.7 Running `espcoredump.py`

Generic command syntax: `espcoredump.py [options] command [args]`

Script Options

–**chip {auto,esp32,esp32s2,esp32s3,esp32c2,esp32c3}** Target chip type. Default value is "auto"

--port PORT, -p PORT Serial port device. Either "chip" or "port" need to be specified to determine the port when you have multi-target connected at the same time.

--baud BAUD, -b BAUD Serial port baud rate used when flashing/reading

--gdb-timeout-sec GDB_TIMEOUT_SEC Overwrite the default internal delay for gdb responses

Commands **dbg_corefile** Starts GDB debugging session with specified corefile

info_corefile Print core dump info from file

Command Arguments

--debug DEBUG, -d DEBUG Log level (0..3)

--gdb GDB, -g GDB Path to gdb

--core CORE, -c CORE Path to core dump file (if skipped core dump will be read from flash)

–**core-format {b64,elf,raw}, -t {b64,elf,raw}** File specified with "-c" is an ELF ("elf"), raw (raw) or base64-encoded (b64) binary

--off OFF, -o OFF Offset of coredump partition in flash (type "idf.py partition-table" to see).

--save-core SAVE_CORE, -s SAVE_CORE Save core to file. Otherwise temporary core file will be deleted. Does not work with "-c"

--rom-elf ROM_ELF, -r ROM_ELF Path to ROM ELF file. Will use "<target>_rom.elf" if not specified

--print-mem, -m Print memory dump. Only valid when info_corefile.

<prog> Path to program ELF file.

Related Documents

Anatomy of core dump image Core dump component can be configured to use old legacy binary format or the new ELF one. The ELF format is recommended for new designs. It provides more information about the CPU and memory state of a program at the moment when panic handler is entered. The memory state embeds a snapshot of all tasks mapped in the memory space of the program. The CPU state contains register values when the core dump has been generated. Core dump file uses a subset of the ELF structures to register these information. Loadable ELF segments are used for the memory state of the process while ELF notes (ELF.PT_NOTE) are used for process metadata (pid, registers, signal, ⋯). Especially, the CPU status is stored in a note with a special name and type (`CORE`, `NT_PRSTATUS type`).

Here is an overview of coredump layout:

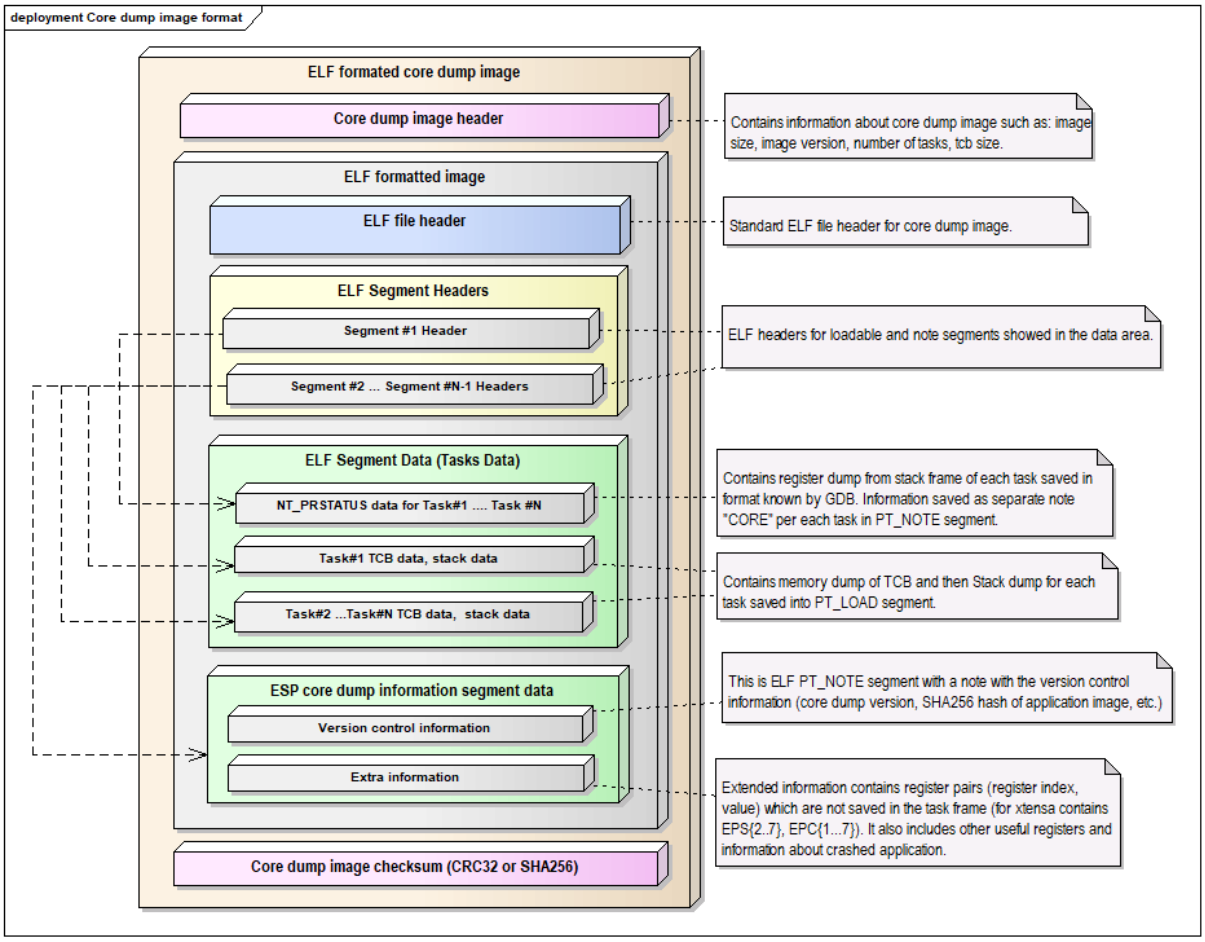


Fig. 4: Core dump ELF image format

Note: The format of image file showed on the above pictures represents current version of image and can be changed in future releases.

Overview of implementation The figure below describes some basic aspects related to implementation of core dump:

Note: The diagram above hide some details and represents current implementation of the core dump and can be changed later.

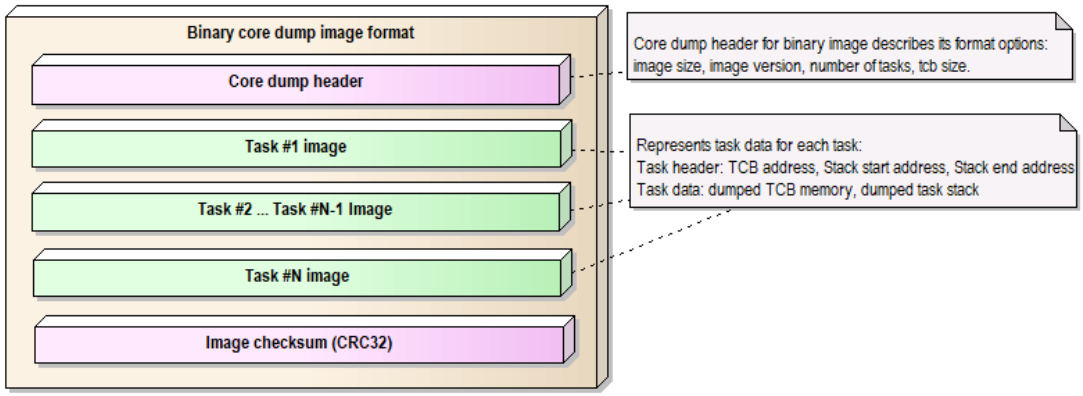


Fig. 5: Core dump binary image format

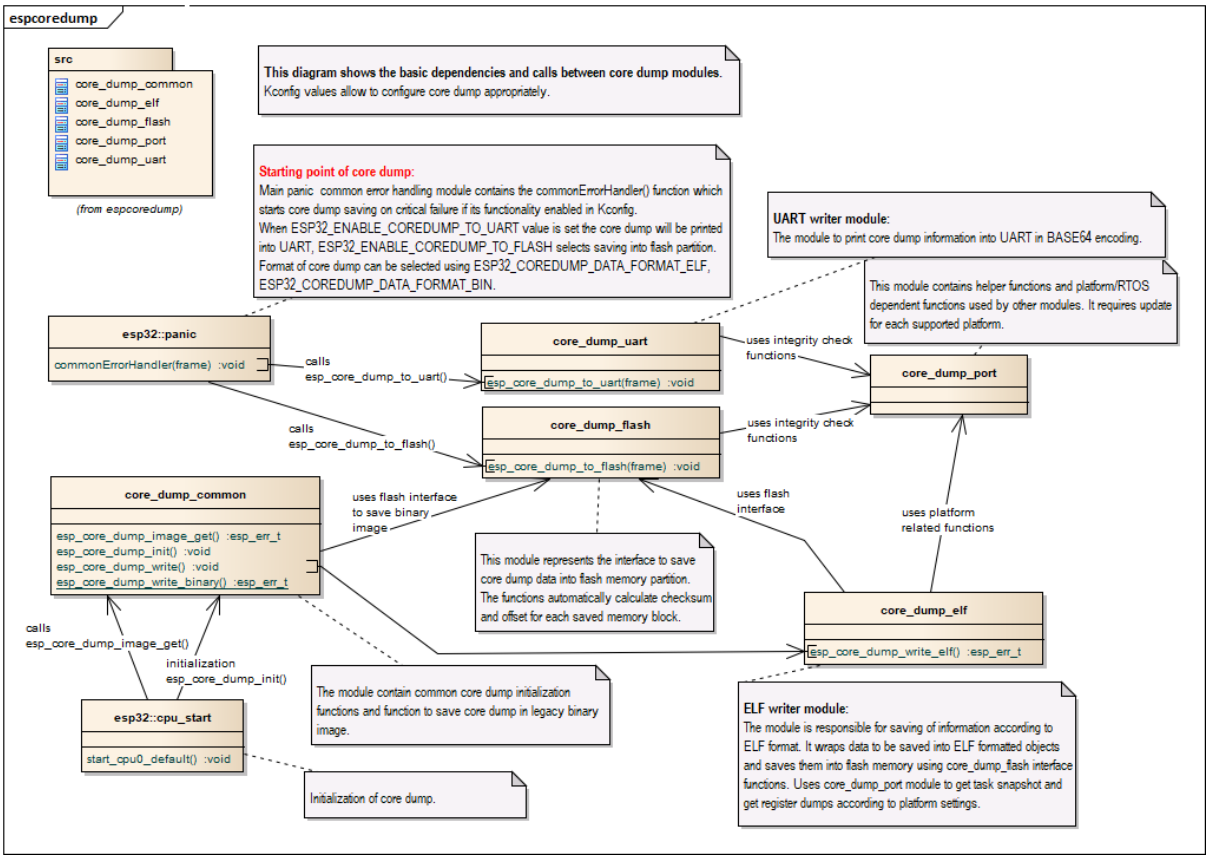


Fig. 6: Core dump implementation overview

4.7 Deep Sleep Wake Stubs

ESP32 supports running a "deep sleep wake stub" when coming out of deep sleep. This function runs immediately as soon as the chip wakes up - before any normal initialisation, bootloader, or ESP-IDF code has run. After the wake stub runs, the SoC can go back to sleep or continue to start ESP-IDF normally.

Deep sleep wake stub code is loaded into "RTC Fast Memory" and any data which it uses must also be loaded into RTC memory. RTC memory regions hold their contents during deep sleep.

4.7.1 Rules for Wake Stubs

Wake stub code must be carefully written:

- As the SoC has freshly woken from sleep, most of the peripherals are in reset states. The SPI flash is unmapped.
- The wake stub code can only call functions implemented in ROM or loaded into RTC Fast Memory (see below.)
- The wake stub code can only access data loaded in RTC memory. All other RAM will be unintiailised and have random contents. The wake stub can use other RAM for temporary storage, but the contents will be overwritten when the SoC goes back to sleep or starts ESP-IDF.
- RTC memory must include any read-only data (.rodata) used by the stub.
- Data in RTC memory is initialised whenever the SoC restarts, except when waking from deep sleep. When waking from deep sleep, the values which were present before going to sleep are kept.
- Wake stub code is a part of the main esp-idf app. During normal running of esp-idf, functions can call the wake stub functions or access RTC memory. It is as if these were regular parts of the app.

4.7.2 Implementing A Stub

The wake stub in esp-idf is called `esp_wake_deep_sleep()`. This function runs whenever the SoC wakes from deep sleep. There is a default version of this function provided in esp-idf, but the default function is weak-linked so if your app contains a function named `esp_wake_deep_sleep()` then this will override the default.

If supplying a custom wake stub, the first thing it does should be to call `esp_default_wake_deep_sleep()`.

It is not necessary to implement `esp_wake_deep_sleep()` in your app in order to use deep sleep. It is only necessary if you want to have special behaviour immediately on wake.

If you want to swap between different deep sleep stubs at runtime, it is also possible to do this by calling the `esp_set_deep_sleep_wake_stub()` function. This is not necessary if you only use the default `esp_wake_deep_sleep()` function.

All of these functions are declared in the `esp_sleep.h` header under components/esp32.

4.7.3 Loading Code Into RTC Memory

Wake stub code must be resident in RTC Fast Memory. This can be done in one of two ways.

The first way is to use the `RTC_IRAM_ATTR` attribute to place a function into RTC memory:

```
void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    // Add additional functionality here
}
```

The second way is to place the function into any source file whose name starts with `rtc_wake_stub`. Files names `rtc_wake_stub*` have their contents automatically put into RTC memory by the linker.

The first way is simpler for very short and simple code, or for source files where you want to mix "normal" and "RTC" code. The second way is simpler when you want to write longer pieces of code for RTC memory.

4.7.4 Loading Data Into RTC Memory

Data used by stub code must be resident in RTC memory.

The data can be placed in RTC Fast memory or in RTC Slow memory which is also used by the ULP.

Specifying this data can be done in one of two ways:

The first way is to use the `RTC_DATA_ATTR` and `RTC_RODATA_ATTR` to specify any data (writeable or read-only, respectively) which should be loaded into RTC memory:

```
RTC_DATA_ATTR int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    static RTC_RODATA_ATTR const char fmt_str[] = "Wake count %d\n";
    esp_rom_printf(fmt_str, wake_count++);
}
```

The RTC memory area where this data will be placed can be configured via menuconfig option named `CONFIG_ESP32_RTCDATA_IN_FAST_MEM`. This option allows to keep slow memory area for ULP programs and once it is enabled the data marked with `RTC_DATA_ATTR` and `RTC_RODATA_ATTR` are placed in the RTC fast memory segment otherwise it goes to RTC slow memory (default option). This option depends on the `CONFIG_FREERTOS_UNICORE` because RTC fast memory can be accessed only by PRO_CPU.

The attributes `RTC_FAST_ATTR` and `RTC_SLOW_ATTR` can be used to specify data that will be force placed into RTC_FAST and RTC_SLOW memory respectively. Any access to data marked with `RTC_FAST_ATTR` is allowed by PRO_CPU only and it is responsibility of user to make sure about it.

Unfortunately, any string constants used in this way must be declared as arrays and marked with RTC_RODATA_ATTR, as shown in the example above.

The second way is to place the data into any source file whose name starts with `rtc_wake_stub`.

For example, the equivalent example in `rtc_wake_stub_counter.c`:

```
int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    esp_rom_printf("Wake count %d\n", wake_count++);
}
```

The second way is a better option if you need to use strings, or write other more complex code.

To reduce wake-up time use the *CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP* Kconfig option, see more information in *Fast boot from Deep Sleep*.

4.7.5 CRC Check For Wake Stubs

During deep sleep, all RTC Fast memory areas will be validated with CRC. When ESP32 wakes up from deep sleep, the RTC fast memory will be validated with CRC again. If the validation passes, the wake stubs code will be executed. Otherwise, the normal initialization, bootloader and esp-idf codes will be executed.

Note: When the *CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP* option is enabled, all the RTC fast memory except the wake stubs area is added to the heap.

4.7.6 Example

ESP-IDF provides an example to show how to implement the Deep-sleep wake stub.

- system/deep_sleep_wake_stub

4.8 Error Handling

4.8.1 Overview

Identifying and handling run-time errors is important for developing robust applications. There can be multiple kinds of run-time errors:

- Recoverable errors:
  - Errors indicated by functions through return values (error codes)
  - C++ exceptions, thrown using `throw` keyword
- Unrecoverable (fatal) errors:
  - Failed assertions (using `assert` macro and equivalent methods, see *Assertions*) and `abort()` calls.
  - CPU exceptions: access to protected regions of memory, illegal instruction, etc.
  - System level checks: watchdog timeout, cache access error, stack overflow, stack smashing, heap corruption, etc.

This guide explains ESP-IDF error handling mechanisms related to recoverable errors, and provides some common error handling patterns.

For instructions on diagnosing unrecoverable errors, see *Fatal Errors*.

4.8.2 Error codes

The majority of ESP-IDF-specific functions use *esp_err_t* type to return error codes. *esp_err_t* is a signed integer type. Success (no error) is indicated with `ESP_OK` code, which is defined as zero.

Various ESP-IDF header files define possible error codes using preprocessor defines. Usually these defines start with `ESP_ERR_` prefix. Common error codes for generic failures (out of memory, timeout, invalid argument, etc.) are defined in `esp_err.h` file. Various components in ESP-IDF may define additional error codes for specific situations.

For the complete list of error codes, see *Error Code Reference*.

4.8.3 Converting error codes to error messages

For each error code defined in ESP-IDF components, *esp_err_t* value can be converted to an error code name using *esp_err_to_name()* or *esp_err_to_name_r()* functions. For example, passing `0x101` to *esp_err_to_name()* will return "ESP_ERR_NO_MEM" string. Such strings can be used in log output to make it easier to understand which error has happened.

Additionally, *esp_err_to_name_r()* function will attempt to interpret the error code as a standard POSIX error code, if no matching `ESP_ERR_` value is found. This is done using `strerror_r` function. POSIX error codes (such as `ENOENT`, `ENOMEM`) are defined in `errno.h` and are typically obtained from `errno` variable. In ESP-IDF this variable is thread-local: multiple FreeRTOS tasks have their own copies of `errno`. Functions which set `errno` only modify its value for the task they run in.

This feature is enabled by default, but can be disabled to reduce application binary size. See *CONFIG_ESP_ERR_TO_NAME_LOOKUP*. When this feature is disabled, *esp_err_to_name()* and *esp_err_to_name_r()* are still defined and can be called. In this case, *esp_err_to_name()* will return `UNKNOWN ERROR`, and *esp_err_to_name_r()* will return `Unknown error 0xXXXX(YYYYY)`, where `0xXXXX` and `YYYYY` are the hexadecimal and decimal representations of the error code, respectively.

4.8.4 ESP_ERROR_CHECK macro

ESP_ERROR_CHECK macro serves similar purpose as `assert`, except that it checks *esp_err_t* value rather than a `bool` condition. If the argument of *ESP_ERROR_CHECK* is not equal *ESP_OK*, then an error message is printed on the console, and `abort()` is called.

Error message will typically look like this:

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf

file: "/Users/user/esp/example/main/main.c" line 20
func: app_main
expression: sdmmc_card_init(host, &card)

Backtrace: 0x40086e7c:0x3ffb4ff0 0x40087328:0x3ffb5010 0x400d1fdf:0x3ffb5030␣
↪0x400d0816:0x3ffb5050
```

Note: If *IDF monitor* is used, addresses in the backtrace will be converted to file names and line numbers.

- The first line mentions the error code as a hexadecimal value, and the identifier used for this error in source code. The latter depends on *CONFIG_ESP_ERR_TO_NAME_LOOKUP* option being set. Address in the program where error has occured is printed as well.
- Subsequent lines show the location in the program where *ESP_ERROR_CHECK* macro was called, and the expression which was passed to the macro as an argument.
- Finally, backtrace is printed. This is part of panic handler output common to all fatal errors. See *Fatal Errors* for more information about the backtrace.

4.8.5 ESP_ERROR_CHECK_WITHOUT_ABORT macro

ESP_ERROR_CHECK_WITHOUT_ABORT macro serves similar purpose as `ESP_ERROR_CHECK`, except that it won't call `abort()`.

4.8.6 ESP_RETURN_ON_ERROR macro

ESP_RETURN_ON_ERROR macro checks the error code, if the error code is not equal *ESP_OK*, it prints the message and returns.

4.8.7 ESP_GOTO_ON_ERROR macro

ESP_GOTO_ON_ERROR macro checks the error code, if the error code is not equal *ESP_OK*, it prints the message, sets the local variable *ret* to the code, and then exits by jumping to *goto_tag*.

4.8.8 ESP_RETURN_ON_FALSE macro

ESP_RETURN_ON_FALSE macro checks the condition, if the condition is not equal *true*, it prints the message and returns with the supplied *err_code*.

4.8.9 ESP_GOTO_ON_FALSE macro

ESP_GOTO_ON_FALSE macro checks the condition, if the condition is not equal *true*, it prints the message, sets the local variable *ret* to the supplied *err_code*, and then exits by jumping to *goto_tag*.

4.8.10 CHECK MACROS Examples

Some examples:

```
static const char* TAG = "Test";

esp_err_t test_func(void)
{
    esp_err_t ret = ESP_OK;

    ESP_ERROR_CHECK(x);                                          // err message␣
↪printed if `x` is not `ESP_OK`, and then `abort()`.
    ESP_ERROR_CHECK_WITHOUT_ABORT(x);                            // err message␣
↪printed if `x` is not `ESP_OK`, without `abort()`.
    ESP_RETURN_ON_ERROR(x, TAG, "fail reason 1");                // err message␣
↪printed if `x` is not `ESP_OK`, and then function returns with code `x`.
    ESP_GOTO_ON_ERROR(x, err, TAG, "fail reason 2");             // err message␣
↪printed if `x` is not `ESP_OK`, `ret` is set to `x`, and then jumps to `err`.
    ESP_RETURN_ON_FALSE(a, err_code, TAG, "fail reason 3");      // err message␣
↪printed if `a` is not `true`, and then function returns with code `err_code`.
    ESP_GOTO_ON_FALSE(a, err_code, err, TAG, "fail reason 4");   // err message␣
↪printed if `a` is not `true`, `ret` is set to `err_code`, and then jumps to␣
↪`err`.

err:
    // clean up
    return ret;
}
```

Note: If the option *CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT* in Kconfig is enabled, the err message will be discarded, while the other action works as is.

The `ESP_RETURN_XX` and `ESP_GOTO_xx` macros can't be called from ISR. While there are `xx_ISR` versions for each of them, e.g., *ESP_RETURN_ON_ERROR_ISR*, these macros could be used in ISR.

4.8.11 Error handling patterns

1. Attempt to recover. Depending on the situation, we may try the following methods:
   - retry the call after some time;
   - attempt to de-initialize the driver and re-initialize it again;
   - fix the error condition using an out-of-band mechanism (e.g reset an external peripheral which is not responding).

   Example:

   ```
   esp_err_t err;
   do {
       err = sdio_slave_send_queue(addr, len, arg, timeout);
       // keep retrying while the sending queue is full
   } while (err == ESP_ERR_TIMEOUT);
   if (err != ESP_OK) {
       // handle other errors
   }
   ```

2. Propagate the error to the caller. In some middleware components this means that a function must exit with the same error code, making sure any resource allocations are rolled back.
   Example:

   ```
   sdmmc_card_t* card = calloc(1, sizeof(sdmmc_card_t));
   if (card == NULL) {
   ```

(continues on next page)

(continued from previous page)

```
    return ESP_ERR_NO_MEM;
}
esp_err_t err = sdmmc_card_init(host, &card);
if (err != ESP_OK) {
    // Clean up
    free(card);
    // Propagate the error to the upper layer (e.g. to notify the user).
    // Alternatively, application can define and return custom error code.
    return err;
}
```

3. Convert into unrecoverable error, for example using `ESP_ERROR_CHECK`. See *ESP_ERROR_CHECK macro* section for details.
   Terminating the application in case of an error is usually undesirable behavior for middleware components, but is sometimes acceptable at application level.
   Many ESP-IDF examples use `ESP_ERROR_CHECK` to handle errors from various APIs. This is not the best practice for applications, and is done to make example code more concise.
   Example:

```
ESP_ERROR_CHECK(spi_bus_initialize(host, bus_config, dma_chan));
```

4.8.12 C++ Exceptions

Support for C++ Exceptions in ESP-IDF is disabled by default, but can be enabled using *CONFIG_COMPILER_CXX_EXCEPTIONS* option.

Enabling exception handling normally increases application binary size by a few KB. Additionally it may be necessary to reserve some amount of RAM for exception emergency pool. Memory from this pool will be used if it is not possible to allocate exception object from the heap. Amount of memory in the emergency pool can be set using *CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE* variable.

If an exception is thrown, but there is no `catch` block, the program will be terminated by `abort` function, and backtrace will be printed. See *Fatal Errors* for more information about backtraces.

See cxx/exceptions for an example of C++ exception handling.

4.9 ESP-BLE-MESH

Bluetooth® mesh networking enables many-to-many (m:m) device communications and is optimized for creating large-scale device networks.

Devices may relay data to other devices not in direct radio range of the originating device. In this way, mesh networks can span very large physical areas and contain large numbers of devices. It is ideally suited for building automation, sensor networks, and other IoT solutions where tens, hundreds, or thousands of devices need to reliably and securely communicate with one another.

Bluetooth mesh is not a wireless communications technology, but a networking technology. This technology is dependent upon Bluetooth Low Energy (BLE) - a wireless communications protocol stack.

Built on top of Zephyr Bluetooth Mesh stack, the ESP-BLE-MESH implementation supports device provisioning and node control. It also supports such node features as Proxy, Relay, Low power and Friend.

Please see the *ESP-BLE-MESH Architecture* for information about the implementation of ESP-BLE-MESH architecture and *ESP-BLE-MESH API Reference* for information about respective API.

ESP-BLE-MESH is implemented and certified based on the latest Mesh Profile v1.0.1, users can refer here for the certification details of ESP-BLE-MESH.

Note: If you are looking for Wi-Fi based implementation of mesh for ESP32, please check another product by Espressif called ESP-WIFI-MESH. For more information and documentation see *ESP-WIFI-MESH*.

4.9.1 Getting Started with ESP-BLE-MESH

This section is intended to help you get started with ESP-BLE-MESH for the hardware based on the ESP32 chip by Espressif.

We are going to demonstrate process of setting and operation of a small ESP-BLE-MESH network of three nodes. This process will cover device provisioning and node configuration, and then sending on/off commands to Generic OnOff Server Models on specific nodes.

If you are new to ESP-IDF, please first set up development environment, compile , flash and run example application following top level ESP-IDF *Get Started* documentation.

What You Need

Hardware:

- Three ESP32 boards, see *options*.
- USB cables to connect the boards.
- Computer configured with ESP-IDF.
- Mobile phone or tablet running Android or iOS.

Software:

- Example application bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server code to load to the ESP32 boards.
- Mobile App: **nRF Mesh** for Android or iOS. Optionally you can use some other Apps:
  - EspBleMesh Android App
  - Silicon Labs Android or iOS App

Installation Step by Step

This is a detailed roadmap to walk you through the installation process.

Step 1. Check Hardware Both ESP32-DevKitC and ESP-WROVER-KIT development boards are supported for ESP-BLE-MESH implementation. You can choose particular board through menuconfig: `idf.py menuconfig > Example Configuration > Board selection for ESP-BLE-MESH`

Note: If you plan to use ESP32-DevKitC, connect a RGB LED to GPIO pins 25, 26 and 27.

Step 2. Configure Software Enter the bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server example directory, run `idf.py menuconfig` to select your board and then run `idf.py build` to compile the example.

Step 3. Upload Application to Nodes After the bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server example is compiled successfully, users can run `idf.py flash` to upload the same generated binary files into each of the three development boards.

Once boards are powered on, the RGB LED on each board should turn **GREEN**.

Step 4. Provision Nodes In this section, we will use the **nRF Mesh Android** App to demonstrate how to provision an unprovisioned device. Users can also get its iOS version from the App Store.

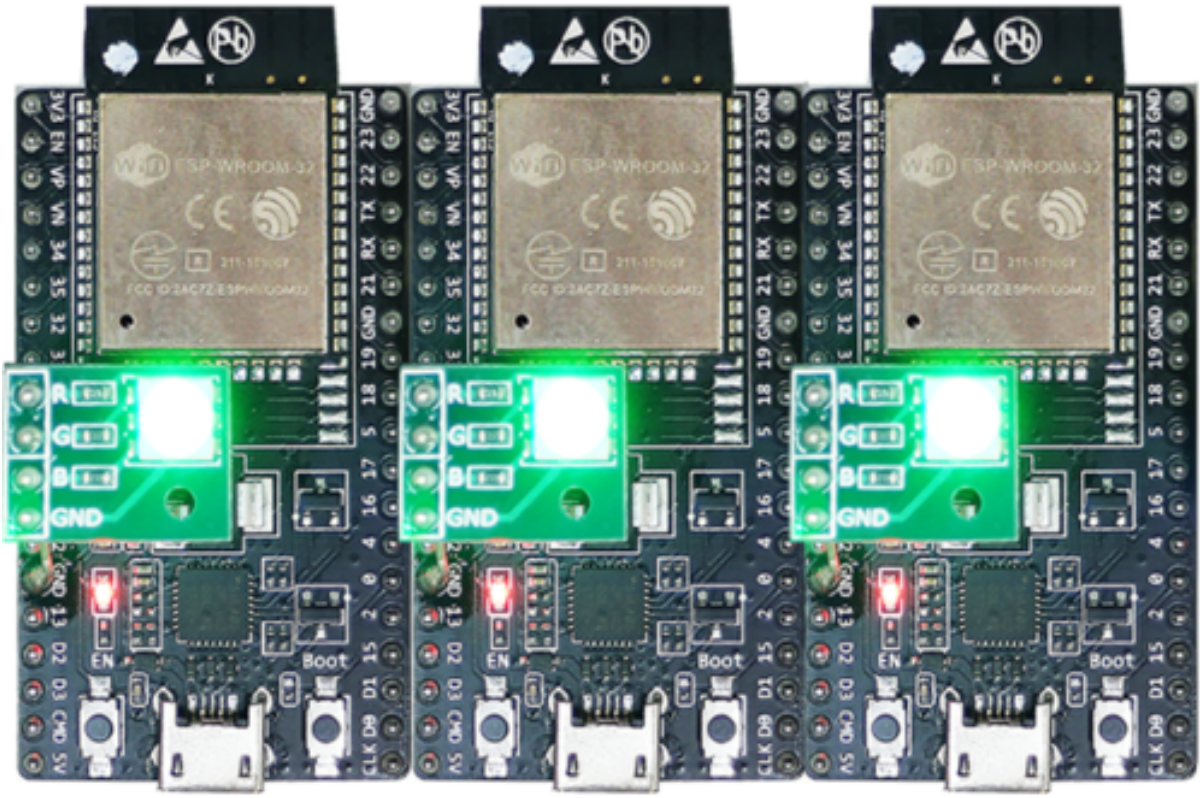


Fig. 7: ESP-BLE-MESH Devices Power On

4.1 Scanner The Scanner is App' s functionality to search for unprovisioned devices in range. Open the App, press **Scanner** at the bottom and the search will start. After a short while we should see three unprovisioned devices displayed.

4.2 Identify Users can select any unprovisioned device, then the App will try to set up a connection with the selected device. After the BLE connection is established successfully (sometimes users need to try multiple times to get connected), and proper ESP-BLE-MESH GATT Service is discovered, users can see the **IDENTIFY** interface button on the screen. The IDENTIFY operation can be used to tell users which device is going to be provisioned.

Note: The IDENTIFY operation also needs some cooperation on the device side, then users can see which device is in the provisioning process. Currently when pressing the **IDENTIFY** interface button, no signs can been seen from the device except from the log on the serial monitor.

After the **IDENTIFY** interface button is pressed, users can see the **PROVISION** interface button.

4.3 Provision Then, the App will try to provision the unprovisioned device. When the device is provisioned successfully, the RGB LED on the board will turn off, and the App will implement the following procedures:

1. Disconnect with the node
2. Try to reconnect with the node
3. Connect successfully and discover ESP-BLE-MESH GATT Service
4. Get Composition Data of the node and add AppKey to it

When all the procedures are finished, the node is configured properly. And after pressing **OK**, users can see that unicast address is assigned, and Composition Data of the node is decoded successfully.

Sometimes in procedure 2, the App may fail to reconnect with the node. In this case, after pressing **OK**, users can see that only unicast address of the node has been assigned, but no Composition Data has been got. Then users need

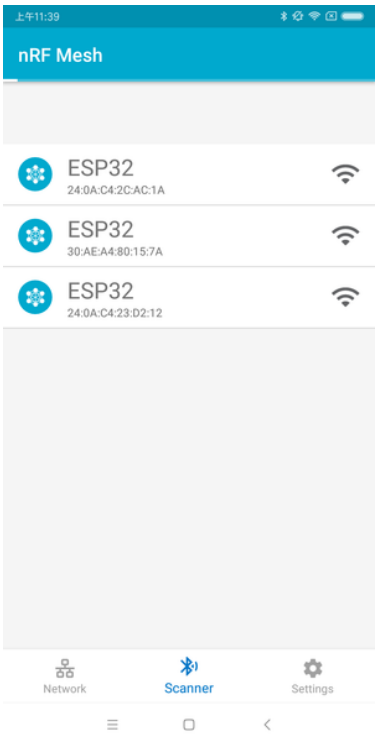


Fig. 8: nRF Mesh - Scanner

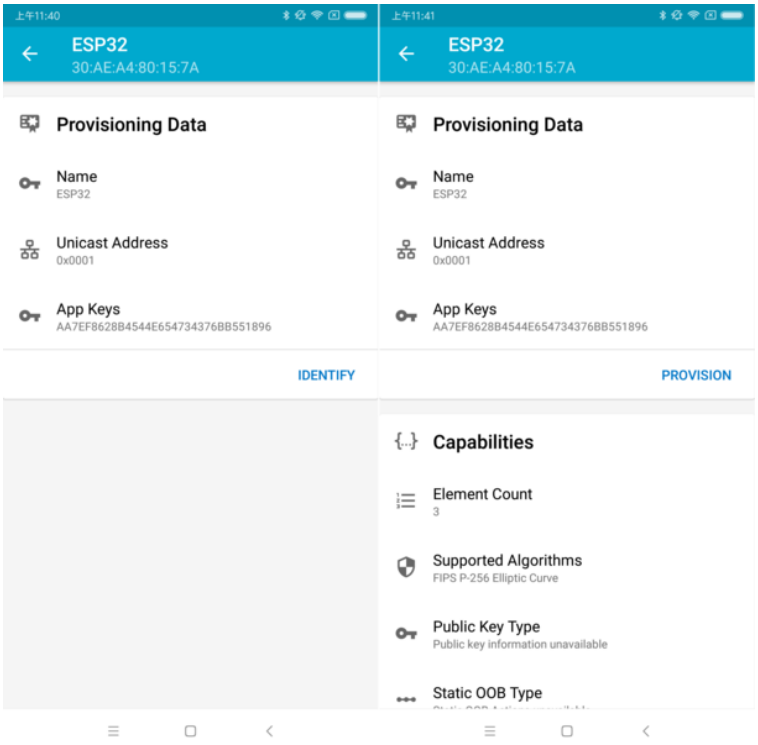


Fig. 9: nRF Mesh - IDENTIFY - PROVISION

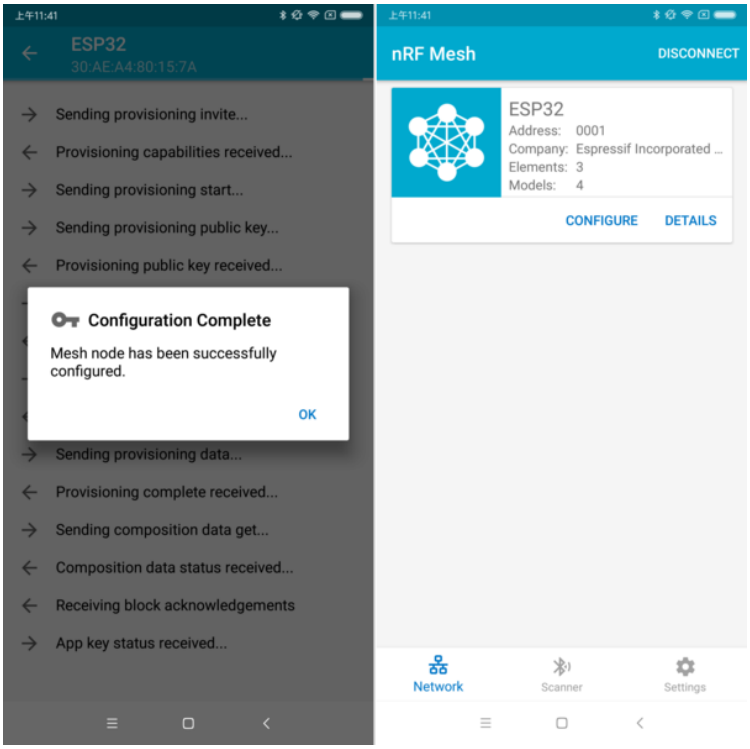


Fig. 10: nRF Mesh - Configuration Complete

to press **CONNECT** on the top right, and the previously provisioned node will be displayed on the screen, and users need to choose it and try to connect with the node.

After connecting successfully, the App will show the interface buttons which can be used to get Composition Data and add AppKey.

If the device is the second or the third one which has been provisioned by the App, and after pressing **CONNECT**, users can see two or three nodes on the screen. In this situation, users can choose any device to connect with, once succeed then go back to the main screen to choose the node which needs to be configured.

Here an example of three devices listed.

- The left picture shows that the third device is provisioned successfully, but the App failed to connect with it. When it tries to reconnect with the third node, three nodes are displayed on the App.
- The right picture shows that after connecting with any node successfully, the App displays the information of the three nodes. Users can see that the App has got the Composition Data of the first and the second nodes, but for the third one, only the unicast address has been assigned to it while the Composition Data is unknown.

4.4 Configuration When provisioning and initial configuration are finished, users can start to configure the node, such as binding AppKey with each model with the elements, setting publication information to it, etc.

Example below shows how to bind AppKey with Generic OnOff Server Model within the Primary Element.

Note: No need to bind AppKey with the Configuration Server Model, since it only uses the DevKey to encrypt messages in the Upper Transport Layer.

Step 5. Operate Network After all the Generic OnOff Server Models within the three elements are bound with proper AppKey, users can use the App to turn on/off the RGB LED.

In the bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server example, the first Generic OnOff Server Model is used to control the **RED** color, the second one is used to control the **GREEN** color and the third one is used to control

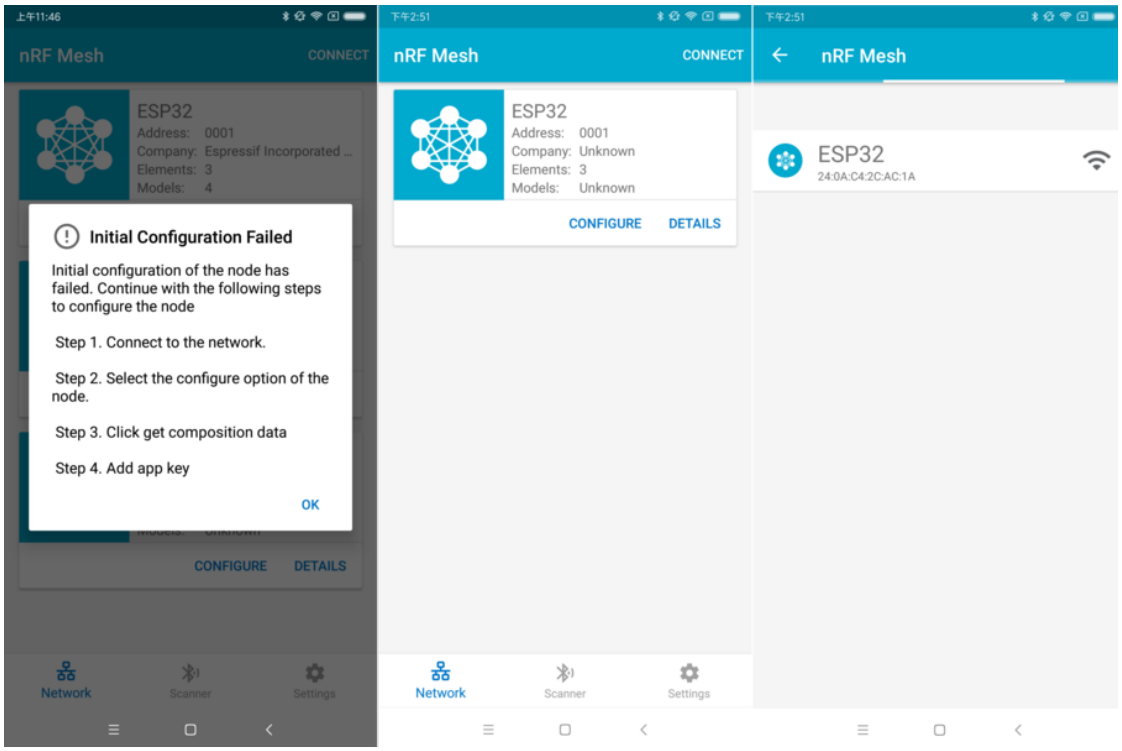


Fig. 11: nRF Mesh - Initial Configuration Failed

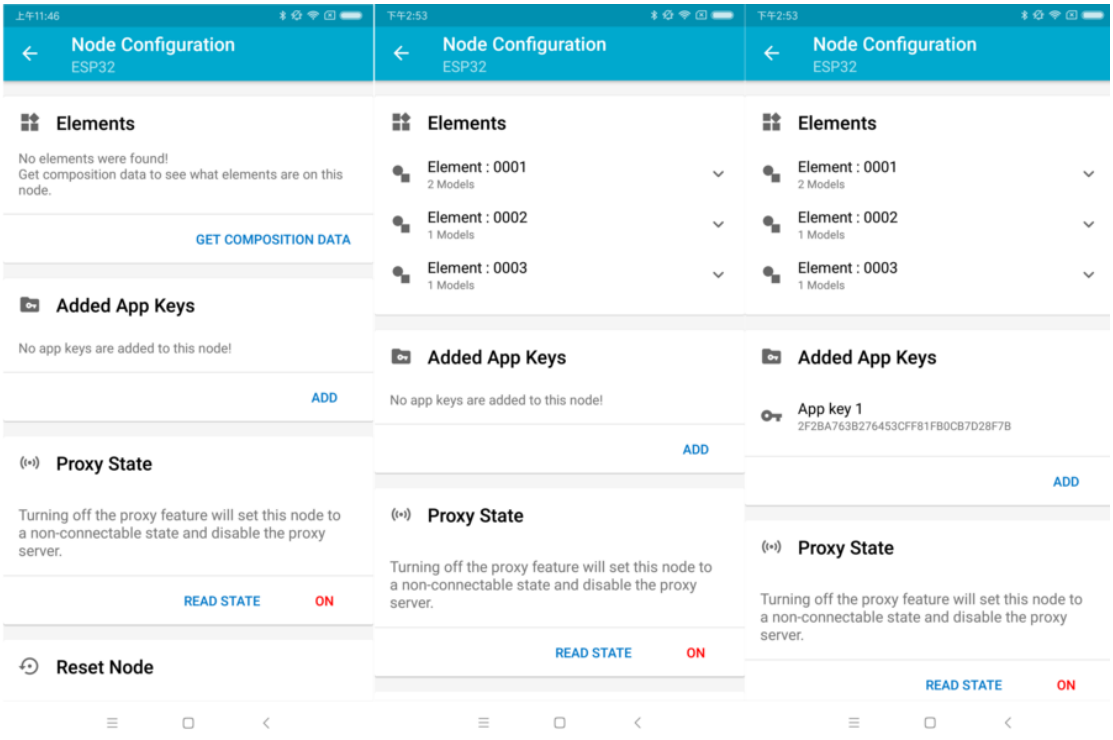


Fig. 12: nRF Mesh - Reconnect - Initial Configuration

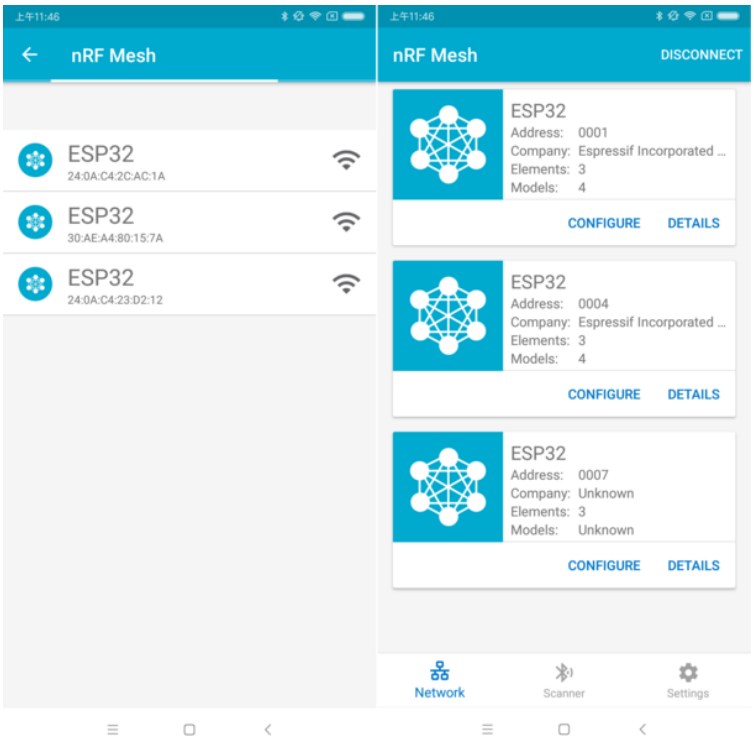


Fig. 13: nRF Mesh - Reconnect - Three Nodes

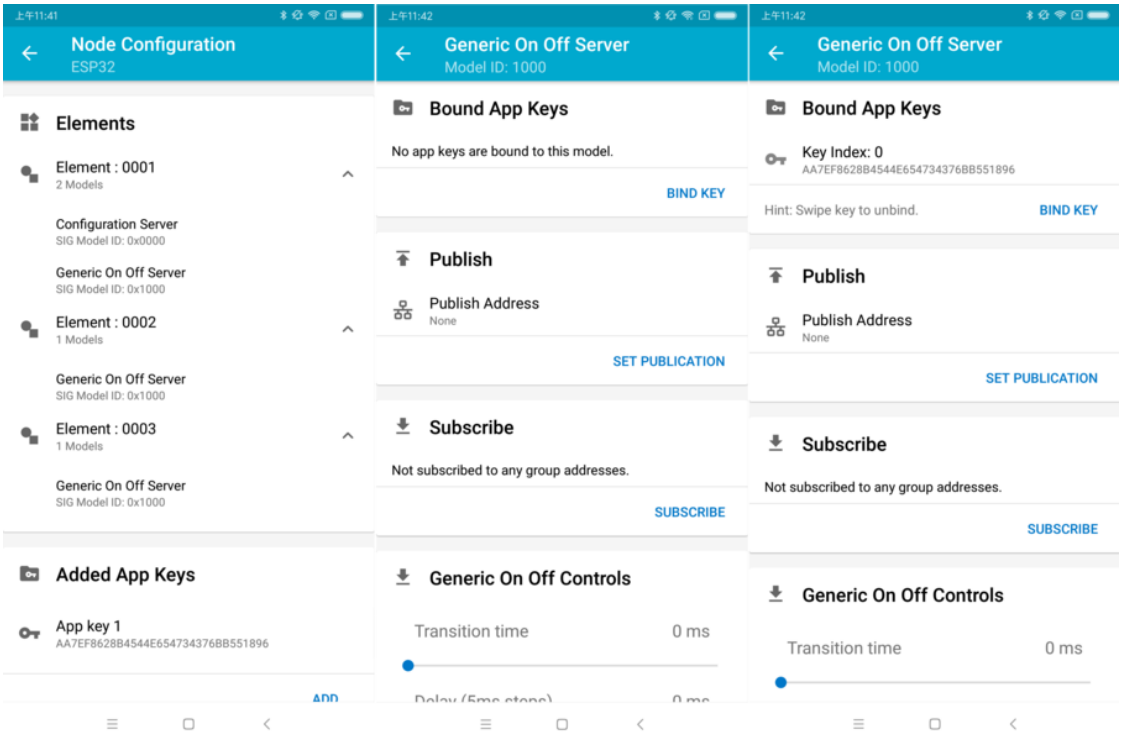


Fig. 14: nRF Mesh - Model Bind AppKey

the **BLUE** color.

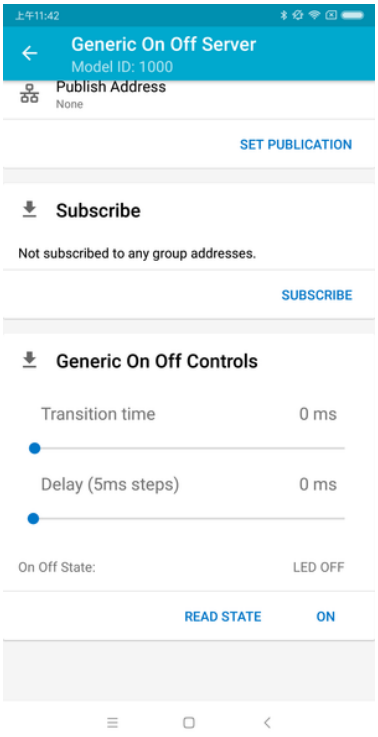


Fig. 15: nRF Mesh - Generic OnOff Control

The following screenshot shows different board with different color on.

Note: For **nRF Mesh** iOS App [version 1.0.4], when the node contains more than one element, the App is not behaving correctly. If users try to turn on/off the second or the third Generic OnOff Server Model, the message sent by the App is destinated to the first Generic OnOff Server Model within the Primary Element.

4.9.2 ESP-BLE-MESH Examples

- ESP-BLE-MESH Node OnOff Server - shows the use of ESP-BLE-MESH as a node having a Configuration Server model and a Generic OnOff Server model. A ESP-BLE-MESH Provisioner can then provision the unprovisioned device and control a RGB LED representing on/off state, see example code .
- ESP-BLE-MESH Node OnOff Client - shows how a Generic OnOff Client model works within a node. The node has a Configuration Server model and a Generic OnOff Client model, see example code .
- ESP-BLE-MESH Provisioner - shows how a device can act as an ESP-BLE-MESH Provisioner to provision devices. The Provisioner has a Configuration Server model, a Configuration Client model and a Generic OnOff Client model, see example code .
- ESP-BLE-MESH Fast Provisioning - Client and Server - this example is used for showing how fast provisioning can be used in order to create a mesh network. It takes no more than 60 seconds to provision 100 devices, see example client code and example server code .
- ESP-BLE-MESH and Wi-Fi Coexistence - an example that demonstrates the Wi-Fi and Bluetooth (BLE/BR/EDR) coexistence feature of ESP32. Simply put, users can use the Wi-Fi while operating Bluetooth, see example code .
- ESP-BLE-MESH Console - an example that implements BLE Mesh basic features. Within this example a node can be scanned and provisioned by Provisioner and reply to get/set message from Provisioner, see example node code .

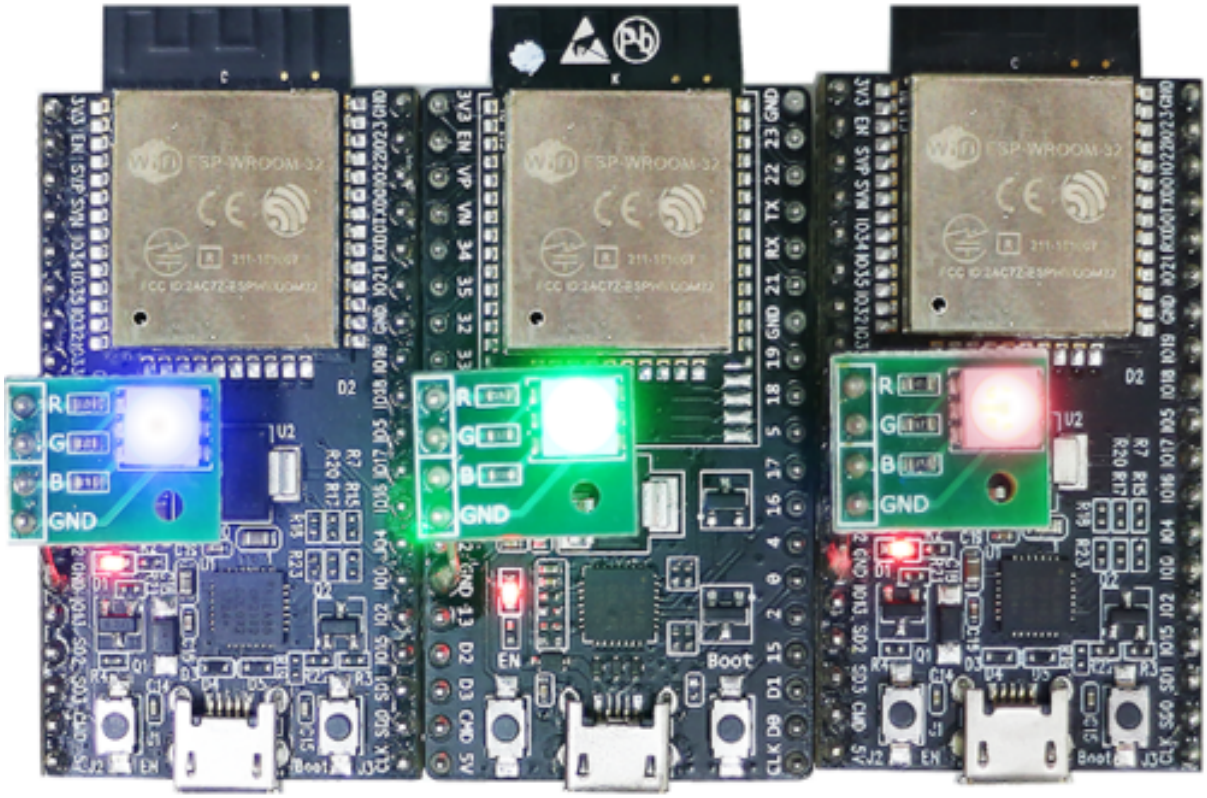


Fig. 16: Three ESP-BLE-MESH Nodes On

4.9.3 ESP-BLE-MESH Demo Videos

- Espressif Fast Provisioning using ESP-BLE-MESH App
- Espressif ESP-BLE-MESH and Wi-Fi Coexistence

4.9.4 ESP-BLE-MESH FAQ

- *1. Provisioner Development*
- *2. Node Development*
- *3. ESP-BLE-MESH and Wi-Fi Coexistence*
- *4. Fast Provisioning*
- *5. Log Help*
- *6. Example Help*
- *7. Others*

4.9.5 Related Documents

ESP-BLE-MESH Feature List

Supported Features

Mesh Core

- **Provisioning: Node Role**
  - PB-ADV and PB-GATT
  - OOB Authentication
- **Provisioning: Provisioner Role**

  - PB-ADV and PB-GATT
  - OOB Authentication
- **Networking**
  - Relay
  - Segmentation and Reassembly
  - Key Refresh Procedure
  - IV Update Procedure
  - Friend
  - Low Power
  - Proxy Server
  - Proxy Client
- **Multiple Client Models Run Simultaneously**
  - Support multiple client models send packets to different nodes simultaneously
  - No blocking between client model and server model
- **NVS Storing**
  - Store provisioning and configuration information of ESP-BLE-MESH Node

Mesh Models

- **Foundation models**
  - Configuration Server model
  - Configuration Client model
  - Health Server model
  - Health Client model
- **Generic client models**
  - Generic OnOff Client
  - Generic Level Client
  - Generic Default Transition Time Client
  - Generic Power OnOff Client
  - Generic Power Level Client
  - Generic Battery Client
  - Generic Location Client
  - Generic Property Client
- **Sensor client models**
  - Sensor Client
- **Time and Scenes client models**
  - Time Client
  - Scene Client
  - Scheduler Client
- **Lighting client models**
  - Light Lightness Client
  - Light CTL Client
  - Light HSL Client
  - Light xyL Client
  - Light LC Client
- **Generic server models**
  - Generic OnOff Server
  - Generic Level Server
  - Generic Default Transition Time Server
  - Generic Power OnOff Server
  - Generic Power OnOff Setup Server
  - Generic Power Level Server
  - Generic Power Level Setup Server
  - Generic Battery Server
  - Generic Location Server
  - Generic Location Setup Server
  - Generic User Property Server
  - Generic Admin Property Server

  - Generic Manufacturer Property Server
  - Generic Client Property Server
- **Sensor server models**
  - Sensor Server
  - Sensor Setup Server
- **Time and Scenes server models**
  - Time Server
  - Time Setup Server
  - Scene Server
  - Scene Setup Server
  - Scheduler Server
  - Scheduler Setup Server
- **Lighting server models**
  - Light Lightness Server
  - Light Lightness Setup Server
  - Light CTL Server
  - Light CTL Temperature Server
  - Light CTL Setup Server
  - Light HSL Server
  - Light HSL Hue Server
  - Light HSL Saturation Server
  - Light HSL Setup Server
  - Light xyL Server
  - Light xyL Setup Server
  - Light LC Server
  - Light LC Setup Server

Mesh Applications

- **ESP-BLE-MESH Node**
  - Tutorial
  - Tutorial
  - Example
- **ESP-BLE-MESH Provisioner**
  - Tutorial
  - Example
- **ESP-BLE-MESH Fast Provisioning**
  - Fast Provisioning Client Model Tutorial
  - Fast Provisioning Server Model Tutorial
  - Example
  - Demo Video
- **ESP-BLE-MESH and Wi-Fi Coexistence**
  - Tutorial
  - Example
  - Demo Video
- **ESP-BLE-MESH Console Commands**
  - Example

Future Release Features

Mesh Core

- Provisioner NVS Storage

Mesh Applications

- Fast OTA

- Friendship

ESP-BLE-MESH Architecture

This document introduces ESP-BLE-MESH architecture overview, ESP-BLE-MESH architecture implementation as well as ESP-BLE-MESH auxiliary routines.

- ESP-BLE-MESH Architecture Overview
  - Describes the five major parts of ESP-BLE-MESH architecture and the functionality of each part.
- ESP-BLE-MESH Architecture Implementation
  - Describes the basic functions of ESP-BLE-MESH files, the correspondence between files and ESP-BLE-MESH architecture, and the interface for calling among files.
- ESP-BLE-MESH Auxiliary Routines
  - Describe the auxiliary routines of ESP-BLE-MESH, such as Mesh network management, Mesh features, etc.

1. ESP-BLE-MESH Architecture Overview Currently ESP-BLE-MESH has implemented most functions of Mesh Profile and all the Client Models defined in Mesh Model specification. Those missing functions/models are under development and will be provided soon. ESP-BLE-MESH architecture has been granted the official Bluetooth certification.

ESP-BLE-MESH architecture includes five key parts:

- `Mesh Protocol Stack`
  - `Mesh Networking` is responsible for processing of messages of ESP-BLE-MESH nodes.
  - `Mesh Provisioning` is responsible for provisioning flow of ESP-BLE-MESH devices.
  - `Mesh Models` is responsible for the implementation of SIG-defined models.
- `Network Management`
  - Implements several network management procedures, including node removal procedure, IV Index recovery procedure, etc.
- `Features`
  - Include several ESP-BLE-MESH features, e.g. Low Power feature, Friend feature, Relay feature, etc.
- `Mesh Bearer Layer`
  - Includes `Advertising Bearer` and `GATT Bearer`. The bearer layer is crucial to ESP-BLE-MESH protocol stack which is built on Bluetooth Low-Energy technology, because the protocol stack must make use of the bearer layer to transmit data via the BLE advertising channel and connection channel.
- `Applications`
  - Based on ESP-BLE-MESH protocol stack and `Mesh Models`.
  - By calling API and handling Event, `Applications` interact with `Mesh Networking` and `Mesh Provisioning` in ESP-BLE-MESH protocol stack, as well as a series of Models provided by `Mesh Models`.

1.1 Mesh Protocol Stack

1.1.1 Mesh Networking `Mesh Networking` in the protocol stack architecture implements the following functions:

- The communication between nodes in the Mesh network.
- Encryption and decryption of messages in the Mesh network.
- Management of Mesh network resources (Network Key, IV Index, etc.).
- Segmentation and reassembly of Mesh network messages.
- Model mapping of messages between different models.
- For more features, please see *ESP-BLE-MESH Feature List*.

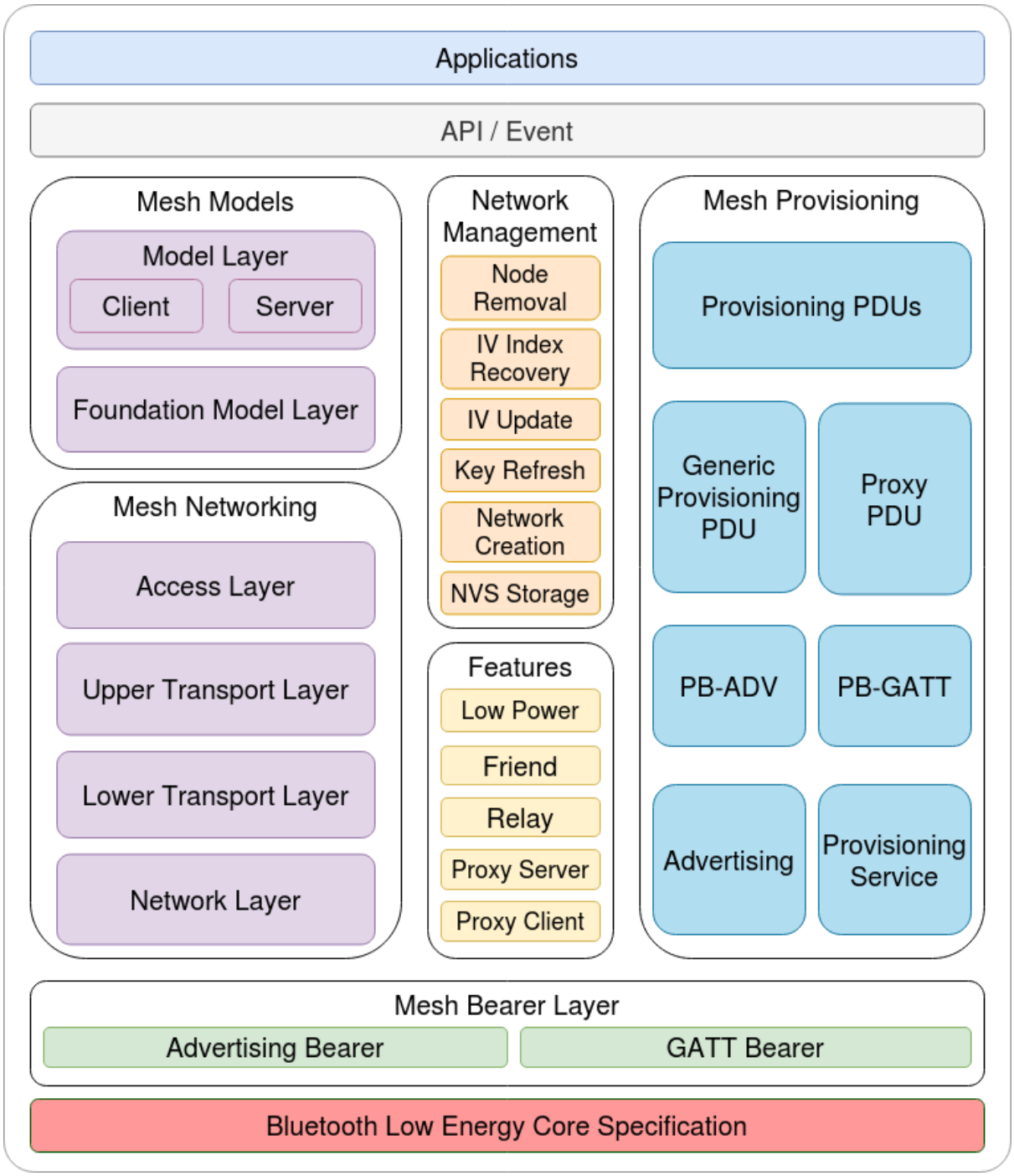


Fig. 17: Figure 1.1 ESP-BLE-MESH Architecture Diagram

The implementation of `Mesh Networking` functions is based on hierarchy structure. Functions of each layer are shown in Table 1.1:

Table 1: Table 1.1 Mesh Networking Architecture Description

Layer	Function
Access Layer	Access Layer not only defines the format of application data, but also defines and controls the encryption and decryption of the data packets conducted by Upper Transport Layer.
Upper Transport Layer	Upper Transport Layer encrypts, decrypts, and authenticates application data to and from the access layer; it also handles special messages called "transport control messages", including messages related to "friendship" and heartbeat messages.
Lower Transport Layer	Lower Transport Layer handles segmentation and reassembly of PDU.
Network Layer	Network Layer defines the address type and format of the network messages, and implements the relay function of the device.

1.1.2 Mesh Provisioning `Mesh Provisioning` in the protocol stack architecture implements the following functions:

- Provisioning of unprovisioned devices.
- Allocation of Mesh network resources (unicast address, IV Index, NetKey, etc.).
- Four authentication methods support during provisioning.
- For more features, please see *ESP-BLE-MESH Feature List*.

The implementation of `Mesh Provisioning` functions is based on hierarchy structure. Functions of each layer are shown in Table 1.2:

Table 2: Table 1.2 Mesh Provisioning Architecture Description

Layer	Function
Provisioning PDUs	Provisioning PDUs from different layers are handled using provisioning protocol.
Generic Provisioning PDU/Proxy PDU	The Provisioning PDUs are transmitted to an unprovisioned device using a Generic Provisioning layer or Proxy protocol layer.
PB-ADV/PB-GATT	These layers define how the Provisioning PDUs are transmitted as transactions that can be segmented and reassembled.
Advertising/Provisioning Service	The provisioning bearers define how sessions are established such that the transactions from the generic provisioning layer can be delivered to a single device.

1.1.3 Mesh Models `Mesh Models` in the protocol stack architecture implements the following functions:

- Configuration Client/Server Models
- Health Client/Server Models
- Generic Client/Server Models
- Sensor Client/Server Models
- Time and Scenes Client/Server Models
- Lighting Client/Server Models

Functions of each layer are shown in Table 1.3:

Table 3: Table 1.3 Mesh Models Architecture Description

Layer	Function
Model Layer	Model Layer implements models used to standardize the operation of typical user scenarios, including Generic Client/Server Models, Sensor Client/Server Models, Time and Scenes Client/Server Models, Lighting Client/Server Models and several vendor models.
Foundation Model Layer	Foundation Model Layer implements models related to ESP-BLE-MESH configuration, management, self diagnosis, etc.

1.2 Mesh Network Management `Network Management` implements the following functions:

- Node removal procedure is used to remove a node from the network.
- IV Index recovery procedure is used to recover a node's IV Index.
- IV update procedure is used to update the nodes' IV Index.
- Key refresh procedure is used to update the nodes' NetKey, AppKey, etc.
- Network creation procedure is used to create a mesh network.
- NVS storage is used to store node's networking information.

1.3 Mesh Features `Features` includes the following options:

- Low Power feature is used to reduce node's power consumption.
- Friend feature is used to store messages for Low Power nodes.
- Relay feature is used to relay/forward Network PDUs received by a node over the advertising bearer.
- Proxy Server/Client are two node roles in proxy protocol, which enable nodes to send and receive Network PDUs, mesh beacons, proxy configuration messages and Provisioning PDUs over a connection-oriented bearer.

1.4 Mesh Bearer Layer `Bearers` in the protocol stack architecture are responsible for passing of data between ESP-BLE-MESH protocol stack and Bluetooth Low Energy Core.

`Bearers` can be taken as a carrier layer based on Bluetooth Low Energy Core, which implements the function of receiving and transmitting data for the ESP-BLE-MESH protocol stack.

Table 4: Table 1.3 Mesh Bearers Description

Layer	Function
GATT Bearer	The GATT Bearer uses the Proxy protocol to transmit and receive `Proxy PDUs` between two devices over a GATT connection.
Advertising Bearer	When using the Advertising Bearer, a mesh packet shall be sent in the Advertising Data of a `Bluetooth Low Energy advertising PDU` using the Mesh Message AD Type.

1.5 Mesh Applications The `Applications` in the protocol stack architecture implement the corresponding functions by calling the API provided by the ESP-BLE-MESH protocol stack and processing the Event reported by the protocol stack. There are some common applications, such as gateway, lighting and etc.

Interaction between application layer `(Applications)` and `API / Event`

- Application layer calls API
  - Call the provisioning-related API for provisioning.
  - Call the model-related API to send messages.
  - Call the device-attributes-related API to get local information about the device.
- Application layer processes Event
  The application layer is designed based on events, which take parameters to the application layer. Events are mainly divided into two categories.
  - **The events completed by calling API.**
    - Such as nodes sending messages.
  - **The events that the protocol stack actively reports to the application layer.**
    - The Event that the protocol stack actively reports.
    - The Event that Model actively reports.
- The event is reported by the callback function registered by the application layer, and the callback function also contains the corresponding processing of the event.

Interaction between `API / Event` and ESP-BLE-MESH protocol stack

- API used by user mainly calls functions provided by `Mesh Networking`, `Mesh Provisioning` and `Mesh Models`.
- The interaction between `API / Event` and the protocol stack does not operate across the hierarchy of the protocol stack. For example, API does not call functions related to `Network Layer`.

2. ESP-BLE-MESH Architecture Implementation The design and implementation of ESP-BLE-MESH architecture is based on layers and modules. In details, Section 2.1 (Mesh Networking Implementation), Section 2.2 (Mesh Provisioning Implementation) and Section 2.3 (Mesh Bearers Implementation) are based on layers, and Section 2.4 (Mesh Models Implementation) is on modules.

- **Layer-based Approach**: With Layer-based approach, the architecture is designed according to the layers specified in the Mesh Profile Specification. Each layer has its unique files which include APIs of this layer and etc. The specific design is shown in Figure 2.1.
- **Module-based Approach**: Every file implements an independent function that can be called by other programs.

The design of ESP-BLE-MESH architecture uses layer-based approach. The sequence of layers which data packets are processed through is fixed, i.e., the processing of packets will form a `message flow`. Thus, we could see flows of messages from the Protocol Stack Interface Diagram in Figure 2.1.

2.1 Mesh Protocol Stack Implementation

2.1.1 Mesh Networking Implementation The list of files and the functions implemented in each file in `Mesh Networking` are shown in Table 2.1:

Table 5: Table 2.1 Mesh Networking File Description :widths: 40 150 :header-rows: 1

File	Functionality
access.c	ESP-BLE-MESH Access Layer
transport.c	ESP-BLE-MESH Lower/Upper Transport Layer
net.c	ESP-BLE-MESH Network Layer
adv.c	A task used to send ESP-BLE-MESH advertising packets, a callback used to handle received advertising packets and APIs used to allocate adv buffers

2.1.2 Mesh Provisioning Implementation The implementation of Mesh Provisioning is divided into two chunks due to the Node/Provisioner coexistence.

Specific files that provide implementation of provisioning of Node are shown in Table 2.2:

Table 6: Table 2.2 Mesh Provisioning (Node) File Description

File	Functionality
prov.c	ESP-BLE-MESH Node provisioning (PB-ADV & PB-GATT)
proxy_server.c	ESP-BLE-MESH Proxy Server related functionalities
beacon.c	APIs used to handle ESP-BLE-MESH Beacons

Specific files that implement functions of Provisioner are shown in Table 2.3:

Table 7: Table 2.3 Mesh Provisioning (Provisioner) File Description

File	Functionality
provisioner_prov.c	ESP-BLE-MESH Provisioner provisioning (PB-ADV & PB-GATT)
proxy_client.c	ESP-BLE-MESH Proxy Client related functionalities
provisioner_main.c	ESP-BLE-MESH Provisioner networking related functionalities

2.1.3 Mesh Models Implementation Mesh Models are used to implement the specific functions of model in nodes. Server model is used to maintain node status. Client model is used to obtain and modify node state.

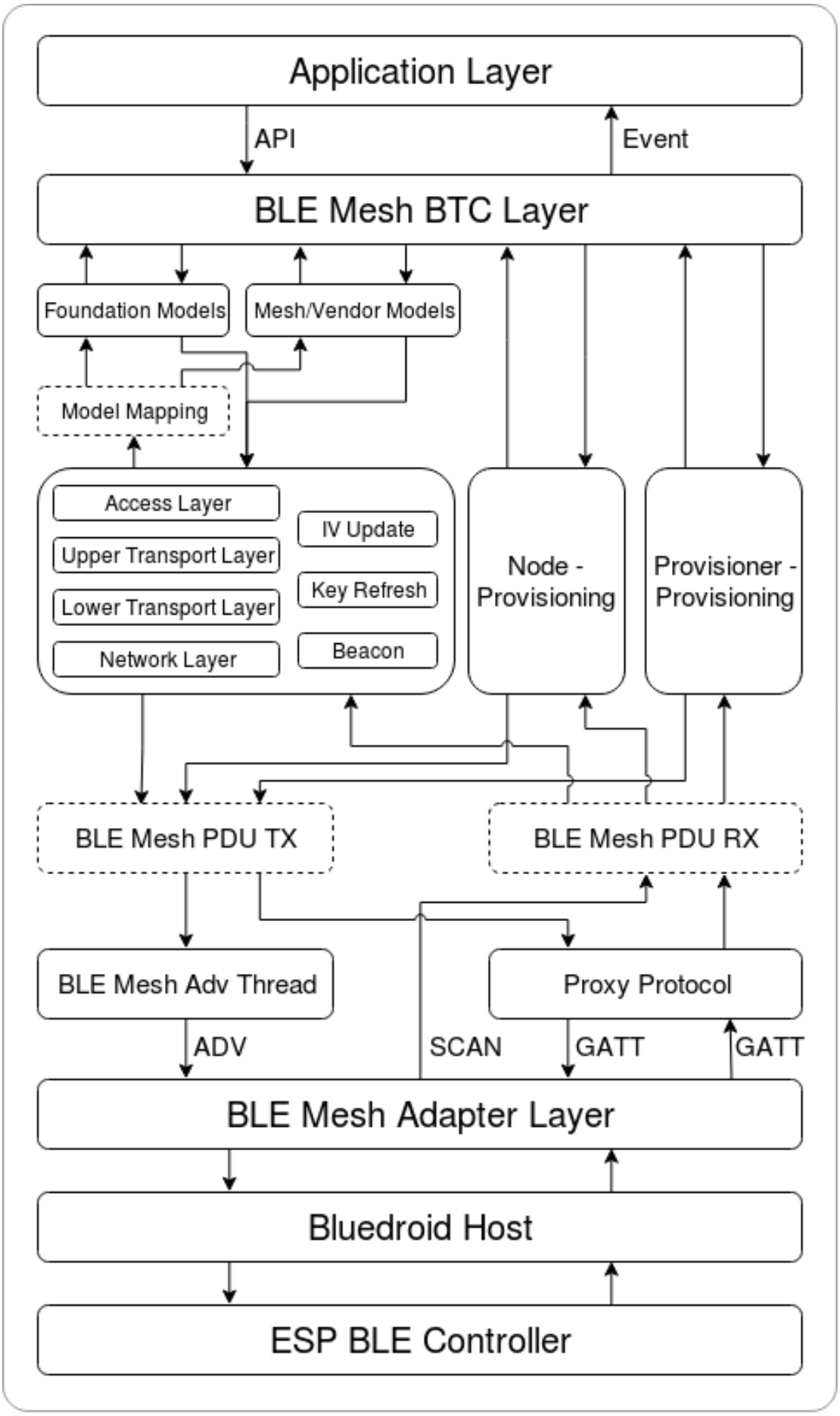


Fig. 18: Figure 2.1 ESP-BLE-MESH Architecture Implementation Diagram

Table 8: Table 2.4 Mesh Models File Description

File	Functionality
cfg_cli.c	Send Configuration Client messages and receive corresponding response messages
cfg_srv.c	Receive Configuration Client messages and send proper response messages
health_cli.c	Send Health Client messages and receive corresponding response messages
health_srv.c	Receive Health Client messages and send proper response messages
client_common.c	ESP-BLE-MESH model related operations
generic_client.c	Send ESP-BLE-MESH Generic Client messages and receive corresponding response messages
lighting_client.c	Send ESP-BLE-MESH Lighting Client messages and receive corresponding response messages
sensor_client.c	Send ESP-BLE-MESH Sensor Client messages and receive corresponding response messages
time_scene_client.c	Send ESP-BLE-MESH Time Scene Client messages and receive corresponding response messages
generic_server.c	Receive ESP-BLE-MESH Generic Client messages and send corresponding response messages
lighting_server.c	Receive ESP-BLE-MESH Lighting Client messages and send corresponding response messages
sensor_server.c	Receive ESP-BLE-MESH Sensor Client messages and send corresponding response messages
time_scene_server.c	Receive ESP-BLE-MESH Time Scene Client messages and send corresponding response messages

2.2 Mesh Bearers Implementation Portability is fully considered in the implementation of Mesh Bearers. When the ESP-BLE-MESH protocol stack is being ported to other platforms, users only need to modify mesh_bearer_adapt.c (example of NimBLE version).

Table 9: Table 2.5 Mesh Bearers File Description

File	Functionality
mesh_bearer_adapt.c	ESP-BLE-MESH Bearer Layer adapter , This file provides the interfaces used to receive and send ESP-BLE-MESH ADV & GATT related packets.

Note: mesh_bearer_adapt.c is the implementation of `Advertising Bearer` and `GATT Bearer` in Mesh Networking framework.

2.3 Mesh Applications Implementation We have provided a series of application examples for customer development, and users can develop products based on *ESP-BLE-MESH Examples*.

3. Auxiliary Routine Auxiliary routine refers to optional functions in the ESP-BLE-MESH protocol stack. The design of the auxiliary routine generally implement the truncation of code through *CONFIG_BLE_MESH*.

3.1 Features

- Low Power
- Friend
- Relay
- Proxy Client/Server

3.2 Network Management

- Node Removal procedure
- IV Index Recovery procedure
- IV Update procedure
- Key Refresh procedure
- Network Creation procedure
- NVS Storage

3.3 Auxiliary Routine Implementation When adopting the design of independent module, the two main factors should be considered:

- The module can not be implemented hierarchically, and it can be completely independent, which means it does not rely on the implementation of other modules.
- The functions in the module will be used repeatedly, so it is reasonable to design it into a module. Independent module is shown in Table 3.1:

Table 10: Table 3.1 Module File Description

File	Functionality
lpn.c	ESP-BLE-MESH Low Power functionality
friend.c	ESP-BLE-MESH Friend functionality
net.c	ESP-BLE-MESH Relay feature, network creation, IV Update procedure, IV Index recovery procedure, Key Refresh procedure related functionalities
proxy_server.c	ESP-BLE-MESH Proxy Server related functionalities
proxy_client.c	ESP-BLE-MESH Proxy Client related functionalities
settings.c	ESP-BLE-MESH NVS storage functionality
main.c	ESP-BLE-MESH stack initialize, stack enable, node removal related functionalities

ESP-BLE-MESH FAQ

This document provides a summary of frequently asked questions about developing with ESP-BLE-MESH, and is divided into seven sections:

- *1. Provisioner Development*
- *2. Node Development*
- *3. ESP-BLE-MESH and Wi-Fi Coexistence*
- *4. Fast Provisioning*
- *5. Log Help*
- *6. Example Help*
- *7. Others*

Users could refer to the sections for quick answer to their questions. This document will be updated based on the feedback collected via various channels.

1. Provisioner Development Generally, a Provisioner is used to provision unprovisioned devices and form a mesh network. And after provisioning, roles of the unprovisioned devices will be changed to those of a node.

1.1 What is the flow for an unprovisioned device to join ESP-BLE-MESH network?

There are two phases for a device to join ESP-BLE-MESH network via a Provisioner, namely, provisioning and configuration.

- The phase of provisioning is to assign unicast address, add NetKey and etc. to a device. By provisioning, the device joins the ESP-BLE-MESH network and its role is changed from an unprovisioned device to a node.

- The phase of configuration is to add AppKeys to the node and bind AppKeys to corresponding models. And some items are optional during configuration, including adding subscription addresses to the node, set publication information, etc. By configuration, the node can actually transmit messages to a Provisioner and receive messages from it.

1.2 If a Provisioner wants to change states of a node, what requirements should be met for a Provisioner?

- Client model that corresponds to server model of the node is required.
- NetKey and AppKey used to encrypt messages shall be owned by both the node and the Provisioner.
- The address owned by the node shall be known, which could be its unicast address or subscription address.

1.3 How can NetKey and AppKey be used?

- NetKey is used for encryption of messages in Network Layer. Nodes with the same NetKey are assumed to be in the same subnet while those with different NetKeys cannot communicate with each other.
- AppKey is used for encryption of messages in Upper Transport Layer. If client model and server model are bound to different AppKeys, the communication cannot be achieved.

1.4 How to generate a NetKey or AppKey for Provisioner? Can we use a fixed NetKey or AppKey?

- The API *esp_ble_mesh_provisioner_add_local_net_key()* can be used to add a NetKey with a fixed or random value.
- The API *esp_ble_mesh_provisioner_add_local_app_key()* can be used to add an AppKey with a fixed or random value.

1.5 Is the unicast address of Provisioner fixed?

The value of `prov_unicast_addr` in *esp_ble_mesh_prov_t* is used to set the unicast address of Provisioner, it can be set only once during initialization and can't be changed afterwards.

1.6 Can the address of Provisioner serve as destination address of the node-reporting-status message ?

The unicast address of Provisioner can be set only once during initialization and can't be changed afterwards. In theory, it can serve as the destination address of the node-reporting-status message, provided that the unicast address of the Provisioner is known by nodes. Nodes can know the unicast address of Provisioner during configuration since Provisioner sends messages to them with its unicast address used as the source address.

Subscription address can also be used. Provisioner subscribes to a group address or virtual address, and nodes send messages to the subscription address.

1.7 Is the unicast address of the node that is firstly provisioned by Provisioner to ESP-BLE-MESH network fixed ?

The value of `prov_start_address` in *esp_ble_mesh_prov_t* is used to set the starting address when the Provisioner provisions unprovisioned devices, i.e. the unicast address of the node it firstly provisioned. It can be set only once during initialization and can't be changed afterwards.

1.8 Is the unicast address of the node that mobile App firstly provisioned fixed?

The App will decide the unicast address, and currently most of them are fixed.

1.9 How to know which unprovisioned device is the Provisioner that is provisioning currently?

The value of `prov_attention` in *esp_ble_mesh_prov_t* is used by Provisioner set to unprovisioned device during provisioning. It can be set only once during initialization and can't be changed afterwards. When the unprovisioned device is joining the mesh network, it can display in a specific way like flashing light to notify Provisioner that it is being provisioned.

1.10 How many ways to authenticate the devices during provisioning? Which way was used in the provided examples ?

There are four authentication methods, i.e. No OOB, Static OOB, Output OOB and Input OOB. In the provided examples, No OOB is used.

1.11 What information can be carried by the advertising packets of the unprovisioned device before provisioning into the network?

- Device UUID
- OOB Info
- URL Hash (optional)

1.12 Can such information be used for device identification?

For example, each unprovisioned device contains a unique Device UUID, which can be used for device identification.

1.13 How is the unicast address assigned when the node provisioned by Provisioner contains multiple elements?

- Provisioner will assign an unicast address for the primary element of the node, and unicast address of the remaining elements are incremented one by one.
- For example: If an unprovisioned device has three elements, i.e. the primary element, the second element and the third element. After provisioning, the primary element address of the node is 0x0002 while the second element address is 0x0003, and the third element address is 0x0004.

1.14 How can Provisioner get and parse the Composition Data of nodes through Configuration Client Model?

- Provisioner can get the Composition Data of nodes using the *Configuration Client Model* API *esp_ble_mesh_config_client_set_state()* with `comp_data_get` in the parameter *esp_ble_mesh_cfg_client_get_state_t* set properly.
- Users can refer to the following code to parse the Composition Data:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>

//test date: 0C001A0001000800030000010501000000800100001003103F002A00
//0C00 1A00 0100 0800 0300 0001 05 01 0000 0080 0100 0010 0310 3F002A00

// CID is 0x000C
// PID is 0x001A
// VID is 0x0001
// CRPL is 0x0008
// Features is 0x0003 - Relay and Friend features.
// Loc is "front" - 0x0100
// NumS is 5
// NumV is 1
// The Bluetooth SIG Models supported are: 0x0000, 0x8000, 0x0001, 0x1000,
↪ 0x1003
```

(continues on next page)

(continued from previous page)

```
// The Vendor Models supported are: Company Identifier 0x003F and Model Identifier 0x002A

typedef struct {
    int16_t cid;
    int16_t pid;
    int16_t vid;
    int16_t crpl;
    int16_t features;
    int16_t all_models;
    uint8_t sig_models;
    uint8_t vnd_models;
} esp_ble_mesh_composition_head;

typedef struct {
    uint16_t model_id;
    uint16_t vendor_id;
} tsModel;

typedef struct {
    // reserve space for up to 20 SIG models
    uint16_t SIG_models[20];
    uint8_t numSIGModels;

    // reserve space for up to 4 vendor models
    tsModel Vendor_models[4];
    uint8_t numVendorModels;
} esp_ble_mesh_composition_decode;

int decode_comp_data(esp_ble_mesh_composition_head *head, esp_ble_mesh_composition_decode *data, uint8_t *mystr, int size)
{
    int pos_sig_base;
    int pos_vnd_base;
    int i;

    memcpy(head, mystr, sizeof(*head));

    if(size < sizeof(*head) + head->sig_models * 2 + head->vnd_models * 4) {
        return -1;
    }

    pos_sig_base = sizeof(*head) - 1;

    for(i = 1; i < head->sig_models * 2; i = i + 2) {
        data->SIG_models[i/2] = mystr[i + pos_sig_base] | (mystr[i + pos_sig_base + 1] << 8);
        printf("%d: %4.4x\n", i/2, data->SIG_models[i/2]);
    }

    pos_vnd_base = head->sig_models * 2 + pos_sig_base;

    for(i = 1; i < head->vnd_models * 2; i = i + 2) {
        data->Vendor_models[i/2].model_id = mystr[i + pos_vnd_base] | (mystr[i + pos_vnd_base + 1] << 8);
        printf("%d: %4.4x\n", i/2, data->Vendor_models[i/2].model_id);

        data->Vendor_models[i/2].vendor_id = mystr[i + pos_vnd_base + 2] | (mystr[i + pos_vnd_base + 3] << 8);
        printf("%d: %4.4x\n", i/2, data->Vendor_models[i/2].vendor_id);
```

(continues on next page)

(continued from previous page)

```
    }

    return 0;
}

void app_main(void)
{
    esp_ble_mesh_composition_head head = {0};
    esp_ble_mesh_composition_decode data = {0};
    uint8_t mystr[] = { 0x0C, 0x00, 0x1A, 0x00,
                        0x01, 0x00, 0x08, 0x00,
                        0x03, 0x00, 0x00, 0x01,
                        0x05, 0x01, 0x00, 0x00,
                        0x00, 0x80, 0x01, 0x00,
                        0x00, 0x10, 0x03, 0x10,
                        0x3F, 0x00, 0x2A, 0x00};
    int ret;

    ret = decode_comp_data(&head, &data, mystr, sizeof(mystr));
    if (ret == -1) {
        printf("decode_comp_data error");
    }
}
```

1.15 How can Provisioner further configure nodes through obtained Composition Data?

Provisioner do the following configuration by calling the *Configuration Client Model* API *esp_ble_mesh_config_client_set_state()*.

- Add AppKey to nodes with `app_key_add` in the parameter *esp_ble_mesh_cfg_client_set_state_t* set properly.
- Add subscription address to the models of nodes with `model_sub_add` in the parameter *esp_ble_mesh_cfg_client_set_state_t* set properly.
- Set publication information to the models of nodes with `model_pub_set` in the parameter *esp_ble_mesh_cfg_client_set_state_t* set properly.

1.16 Can nodes add corresponding configurations for themselves?

This method can be used in special cases like testing period.

- Here is an example to show nodes add new group addresses for their models.

```
esp_err_t example_add_fast_prov_group_address(uint16_t model_id, uint16_t␣
↪group_addr)
{
    const esp_ble_mesh_comp_t *comp = NULL;
    esp_ble_mesh_elem_t *element = NULL;
    esp_ble_mesh_model_t *model = NULL;
    int i, j;

    if (!ESP_BLE_MESH_ADDR_IS_GROUP(group_addr)) {
        return ESP_ERR_INVALID_ARG;
    }

    comp = esp_ble_mesh_get_composition_data();
    if (!comp) {
        return ESP_FAIL;
    }

    for (i = 0; i < comp->element_count; i++) {
```

(continues on next page)

(continued from previous page)

```
        element = &comp->elements[i];
        model = esp_ble_mesh_find_sig_model(element, model_id);
        if (!model) {
            continue;
        }
        for (j = 0; j < ARRAY_SIZE(model->groups); j++) {
            if (model->groups[j] == group_addr) {
                break;
            }
        }
        if (j != ARRAY_SIZE(model->groups)) {
            ESP_LOGW(TAG, "%s: Group address already exists, element␣
↪index: %d", __func__, i);
            continue;
        }
        for (j = 0; j < ARRAY_SIZE(model->groups); j++) {
            if (model->groups[j] == ESP_BLE_MESH_ADDR_UNASSIGNED) {
                model->groups[j] = group_addr;
                break;
            }
        }
        if (j == ARRAY_SIZE(model->groups)) {
            ESP_LOGE(TAG, "%s: Model is full of group addresses, element␣
↪index: %d", __func__, i);
        }
    }

    return ESP_OK;
}
```

Note: When the NVS storage of the node is enabled, group address added and AppKey bound by this method will not be saved in the NVS when the device is powered off currently. These configuration information can only be saved if they are configured by Configuration Client Model.

1.17 How does Provisioner control nodes by grouping?

Generally there are two approaches to implement group control in ESP-BLE-MESH network, group address approach and virtual address approach. And supposing there are 10 devices, i.e., five devices with blue lights and five devices with red lights.

- Method 1: 5 blue lights can subscribe to a group address, 5 red lights subscribe to another one. By sending messages to different group addresses, Provisioner can realize group control.
- Method 2: 5 blue lights can subscribe to a virtual address, 5 red lights subscribe to another one. By sending messages to different virtual addresses, Provisioner can realize group control.

1.18 How does Provisioner add nodes to multiple subnets?

Provisioner can add multiple NetKeys to nodes during configuration, and nodes sharing the same NetKey belong to the same subnet. Provisioner can communicate with nodes on different subnets by using different NetKeys.

1.19 How does Provisioner know if a node in the mesh network is offline?

Node offline is usually defined as: the condition that the node cannot be properly communicated with other nodes in the mesh network due to power failure or some other reasons.

There is no connection between nodes and nodes in the ESP-BLE-MESH network. They communicate with each other through advertising channels.

An example is given here to show how to detect a node is offline by Provisioner.

- The node can periodically send heartbeat messages to Provisioner. And if Provisioner failed to receive heartbeat messages in a certain period, the node is considered to be offline.

Note: The heartbeat message should be designed into a single package (less than 11 bytes), so the transmission and reception of it can be more efficient.

1.20 What operations should be performed when Provisioner removes nodes from the network?

Usually when Provisioner tries to remove node from the mesh network, the procedure includes three main steps:

- Firstly, Provisioner adds the node that need to be removed to the "blacklist".
- Secondly, Provisioner performs the *Key Refresh procedure*.
- Lastly, the node performs node reset procedure, and switches itself to an unprovisioned device.

1.21 In the Key Refresh procedure, how does Provisioner update the Netkey owned by nodes?

- Provisioner updates the NetKey of nodes using the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()` with `net_key_update` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.
- Provisioner updates the AppKey of nodes using the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()` with `app_key_update` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.

1.22 How does Provisioner manage nodes in the mesh network?

ESP-BLE-MESH implements several functions related to basic node management in the example, such as `esp_ble_mesh_store_node_info()`. And ESP-BLE-MESH also provides the API `esp_ble_mesh_provisioner_set_node_name()` which can be used to set the node's local name and the API `esp_ble_mesh_provisioner_get_node_name()` which can be used to get the node's local name.

1.23 What does Provisioner need when trying to control the server model of nodes?

Provisioner must include corresponding client model before controlling the server model of nodes.

Provisioner shall add its local NetKey and AppKey.

- Provisioner add NetKey by calling the API `esp_ble_mesh_provisioner_add_local_net_key()`.
- Provisioner add AppKey by calling the API `esp_ble_mesh_provisioner_add_local_app_key()`.

Provisioner shall configure its own client model.

- Provisioner bind AppKey to its own client model by calling the API `esp_ble_mesh_provisioner_bind_app_key_to_local_model()`.

1.24 How does Provisoner control the server model of nodes?

ESP-BLE-MESH supports all SIG-defined client models. Provisioner can use these client models to control the server models of nodes. And the client models are divided into 6 categories with each category has the corresponding functions.

- Configuration Client Model
  - The API `esp_ble_mesh_config_client_get_state()` can be used to get the `esp_ble_mesh_cfg_client_get_state_t` values of Configuration Server Model.
  - The API `esp_ble_mesh_config_client_set_state()` can be used to set the `esp_ble_mesh_cfg_client_set_state_t` values of Configuration Server Model.

- Health Client Model
  - The API `esp_ble_mesh_health_client_get_state()` can be used to get the `esp_ble_mesh_health_client_get_state_t` values of Health Server Model.
  - The API `esp_ble_mesh_health_client_set_state()` can be used to set the `esp_ble_mesh_health_client_set_state_t` values of Health Server Model.
- Generic Client Models
  - The API `esp_ble_mesh_generic_client_get_state()` can be used to get the `esp_ble_mesh_generic_client_get_state_t` values of Generic Server Models.
  - The API `esp_ble_mesh_generic_client_set_state()` can be used to set the `esp_ble_mesh_generic_client_set_state_t` values of Generic Server Models.
- Lighting Client Models
  - The API `esp_ble_mesh_light_client_get_state()` can be used to get the `esp_ble_mesh_light_client_get_state_t` values of Lighting Server Models.
  - The API `esp_ble_mesh_light_client_set_state()` can be used to set the `esp_ble_mesh_light_client_set_state_t` values of Lighting Server Models.
- Sensor Client Models
  - The API `esp_ble_mesh_sensor_client_get_state()` can be used to get the `esp_ble_mesh_sensor_client_get_state_t` values of Sensor Server Model.
  - The API `esp_ble_mesh_sensor_client_set_state()` can be used to set the `esp_ble_mesh_sensor_client_set_state_t` values of Sensor Server Model.
- **Time and Scenes Client Models**
  - The API `esp_ble_mesh_time_scene_client_get_state()` can be used to get the `esp_ble_mesh_time_scene_client_get_state_t` values of Time and Scenes Server Models.
  - The API `esp_ble_mesh_time_scene_client_set_state()` can be used to set the `esp_ble_mesh_time_scene_client_set_state_t` values of Time and Scenes Server Models.

2. Node Development

2.1 What kind of models are included by nodes?

- In ESP-BLE-MESH, nodes are all composed of a series of models with each model implements some functions of the node.
- Model has two types, client model and server model. Client model can get and set the states of server model.
- Model can also be divided into SIG model and vendor model. All behaviors of SIG models are officially defined while behaviors of vendor models are defined by users.

2.2 Is the format of messages corresponding to each model fixed?

- Messages, which consist of opcode and payload, are divided by opcode.
- The type and the format of the messages corresponding to models are both fixed, which means the messages transmitted between models are fixed.

2.3 Which functions can be used to send messages with the models of nodes?

- For client models, users can use the API `esp_ble_mesh_client_model_send_msg()` to send messages.
- For server models, users can use the API `esp_ble_mesh_server_model_send_msg()` to send messages.
- For publication, users call the API `esp_ble_mesh_model_publish()` to publish messages.

2.4 How to achieve the transmission of messages without packet loss?

Acknowledegd message is needed if users want to transmit messages without packet loss. The default time to wait for corresponding response is set in *CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT*. If the sender waits for the response until the timer expires, the corresponding timeout event would be triggered.

Note: Response timeout can be set in the API *esp_ble_mesh_client_model_send_msg()*. The default value (4 seconds) would be applied if the parameter `msg_timeout` is set to **0**.

2.5 How to send unacknowledged messages?

For client models, users can use the API *esp_ble_mesh_client_model_send_msg()* with the parameter `need_rsp` set to `false` to send unacknowledged messages.

For server models, the messages sent by using the API *esp_ble_mesh_server_model_send_msg()* are always unacknowledged messages.

2.6 How to add subscription address to models?

Subscription address can be added through Configuration Client Model.

2.7 What is the difference between messages sent and published by models?

Messages sent by calling the API *esp_ble_mesh_client_model_send_msg()* or *esp_ble_mesh_server_model_send_msg()* will be sent in the duration determined by the Network Transmit state.

Messages published by calling the API *esp_ble_mesh_model_publish()* will be published determined by the Model Publication state. And the publication of messages is generally periodic or with a fixed number of counts. The publication period and publication count are controlled by the Model Publication state, and can be configured through Configuration Client Model.

2.8 How many bytes can be carried when sending unsegmented messages?

The total payload length (which can be set by users) of unsegmented message is 11 octets, so if the opcode of the message is 2 octets, then the message can carry 9-octets of valid information. For vendor messages, due to the 3-octets opcode, the remaining payload length is 8 octets.

2.9 When should the Relay feature of nodes be enabled?

Users can enable the Relay feature of all nodes when nodes detected in the mesh network are sparse.

For dense mesh network, users can choose to just enable the Relay feature of several nodes.

And users can enable the Relay feature by default if the mesh network size is unknown.

2.10 When should the Proxy feature of node be enabled?

If the unprovisioned device is expected to be provisioned by a phone, then it should enable the Proxy feature since almost all the phones do not support sending ESP-BLE-MESH packets through advertising bearer currently. And after the unprovisioned device is provisioned successfully and becoming a Proxy node, it will communicate with the phone using GATT bearer and using advertising bearer to communicate with other nodes in the mesh network.

2.11 How to use the Proxy filter?

The Proxy filter is used to reduce the number of Network PDUs exchanged between a Proxy Client (e.g. the phone) and a Proxy Server (e.g. the node). And with the Proxy filter, Proxy Client can explicitly request to receive only mesh messages with certain destination addresses from Proxy Server.

2.12 When a message can be relayed by a Relay node?

If a message need to be relayed, the following conditions should be met.

- The message is in the mesh network.
- The message is not sent to the unicast address of the node.
- The value of TTL in the message is greater than 1.

2.13 If a message is segmented into several segments, should the other Relay nodes just relay when one of these segments is received or wait until the message is received completely?

Relay nodes will forward segments when one of them are received rather than keeping waiting until all the segments are received.

2.14 What is the principle of reducing power consumption using Low Power feature?

- When the radio is turned on for listening, the device is consuming energy. When low power feature of the node is enabled, it will turn off its radio in the most of the time.
- And cooperation is needed between low power node and friend node, thus low power node can receive messages at an appropriate or lower frequency without the need to keep listening.
- When there are some new messages for low power node, its friend node will store the messages for it. And low power node can poll friend nodes to see if there are new messages at a fixed interval.

2.15 How to continue the communication on the network after powering-down and powering-up again?

Enable the configuration `Store ESP-BLE-MESH Node configuration persistently` in *menuconfig*.

2.16 How to send out the self-test results of nodes?

It is recommended that nodes can publish its self-test results periodically through Health Server Model.

2.17 How to transmit information between nodes?

One possible application scenario for transmitting information between nodes is that spray nodes would be triggered once smoke alarm detected high smoke concentration. There are two approaches in implementation.

- Approach 1 is that spray node subscribes to a group address. When smoke alarm detects high smoke concentration, it will publish a message whose destination address is the group address which has been subscribed by spray node.
- Approach 2 is that Provisioner can configure the unicast address of spray node to the smoke alarm. When high smoke concentration is detected, smoke alarm can use send messages to the spray node with the spray node ' s unicast address as the destination address.

2.18 Is gateway a must for nodes communication?

- Situation 1: nodes only communicate within the mesh network. In this situation, no gateway is need. ESP-BLE-MESH network is a flooded network, messages in the network have no fixed paths, and nodes can communicate with each other freely.
- Situation 2: if users want to control the nodes remotely, for example turn on some nodes before getting home, then a gateway is needed.

2.19 When will the IV Update procedure be performed?

IV Update procedure would be performed once sequence number of messages sent detected by the bottom layer of node reached a critical value.

2.20 How to perform IV Update procedure?

Nodes can perform IV Update procedure with Secure Network Beacon.

3. ESP-BLE-MESH and Wi-Fi Coexistence

3.1 Which modes does Wi-Fi support when it coexists with ESP-BLE-MESH?

Currently only Wi-Fi station mode supports the coexistence.

3.2 Why is the Wi-Fi throughput so low when Wi-Fi and ESP-BLE-MESH coexist?

The *ESP32-DevKitC* board without PSRAM can run properly but the throughput of it is low since it has no PSRAM. When Bluetooth and Wi-Fi coexist, the throughput of ESP32-DevKitC with PSRAM can be stabilized to more than 1Mbps.

Some configurations in menuconfig shall be enabled to support PSRAM.

- `ESP32-specific --> Support for external,SPI-connected RAM --> Try to allocate memories of Wi-Fi and LWIP...`
- `Bluetooth --> Bluedroid Enable --> BT/BLE will first malloc the memory from the PSRAM`
- `Bluetooth --> Bluedroid Enable --> Use dynamic memory allocation in BT/BLE stack.`
- `Bluetooth --> Bluetooth controller --> BLE full scan feature supported.`
- `Wi-Fi --> Software controls Wi-Fi/Bluetooth coexistence --> Wi-Fi`

4. Fast Provisioning

4.1 Why is fast provisioning needed?

Normally when they are several unprovisioned devices, users can provision them one by one. But when it comes to a large number of unprovisioned devices (e.g. 100), provisioning them one by one will take huge amount of time. With fast provisioning, users can provision 100 unprovisioned devices in about 50 seconds.

4.2 Why EspBleMesh App would wait for a long time during fast provisioning?

After the App provisioned one Proxy node, it will disconnect from the App during fast provisioning, and reconnect with the App when all the nodes are provisioned.

4.3 Why is the number of node addresses displayed in the App is more than that of existing node addresses?

Each time after a fast provisioning process, and before starting a new one, the node addresses in the App should be cleared, otherwise the number of the node address will be incorrect.

4.4 What is the usage of the count value which was input in EspBleMesh App?

The **count** value is provided to the Proxy node which is provisioned by the App so as to determine when to start Proxy advertising in advance.

4.5 When will Configuration Client Model of the node running fast_prov_server example start to work?

Configuration Client Model will start to work after the Temporary Provisioner functionality is enabled.

4.6 Will the Temporary Provisioner functionality be enabled all the time?

After the nodes receive messages used to turn on/off lights, all the nodes will disable its Temporary Provisioner functionality and become nodes.

5. Log Help You can find meaning of errors or warnings when they appear at the bottom of ESP-BLE-MESH stack.

5.1 What is the meaning of warning `ran out of retransmit attempts`?

When the node transmits a segmented message, and due to some reasons, the receiver doesn't receive the complete message. Then the node will retransmit the message. When the retransmission count reaches the maximum number, which is 4 currently, then this warning will appear.

5.2 What is the meaning of warning `Duplicate found in Network Message Cache`?

When the node receives a message, it will compare the message with the ones stored in the network cache. If the same has been found in the cache, which means it has been received before, then the message will be dropped.

5.3 What is the meaning of warning `Incomplete timer expired`?

When the node doesn't receive all the segments of a segmented message during a certain period (e.g. 10 seconds), then the Incomplete timer will expire and this warning will appear.

5.4 What is the meaning of warning `No matching TX context for ack`?

When the node receives a segment ack and it doesn't find any self-send segmented message related with this ack, then this warning will appear.

5.5 What is the meaning of warning `No free slots for new incoming segmented messages`?

When the node has no space for receiving new segmented message, this warning will appear. Users can make the space larger through the configuration *CONFIG_BLE_MESH_RX_SEG_MSG_COUNT*.

5.6 What is the meaning of error `Model not bound to Appkey 0x0000` ?

When the node sends messages with a model and the model has not been bound to the AppKey with AppKey Index 0x000, then this error will appear.

5.7 What is the meaning of error `Busy sending message to DST xxxx`?

This error means client model of the node has transmitted a message to the target node and now is waiting for a response, users can not send messages to the same node with the same unicast address. After the corresponding response is received or timer is expired, then another message can be sent.

6. Example Help

6.1 How are the ESP-BLE-MESH callback functions classified?

- The API *esp_ble_mesh_register_prov_callback()* is used to register callback function used to handle provisioning and networking related events.
- The API *esp_ble_mesh_register_config_client_callback()* is used to register callback function used to handle Configuration Client Model related events.
- The API *esp_ble_mesh_register_config_server_callback()* is used to register callback function used to handle Configuration Server Model related events.

- The API *esp_ble_mesh_register_health_client_callback()* is used to register callback function used to handle Health Client Model related events.
- The API *esp_ble_mesh_register_health_server_callback()* is used to register callback function used to handle Health Server Model related events.
- The API *esp_ble_mesh_register_generic_client_callback()* is used to register callback function used to handle Generic Client Models related events.
- The API *esp_ble_mesh_register_light_client_callback()* is used to register callback function used to handle Lighting Client Models related events.
- The API *esp_ble_mesh_register_sensor_client_callback()* is used to register callback function used to handle Sensor Client Model related events.
- The API *esp_ble_mesh_register_time_scene_client_callback()* is used to register callback function used to handle Time and Scenes Client Models related events.
- The API *esp_ble_mesh_register_custom_model_callback()* is used to register callback function used to handle vendor model and unrealized server models related events.

7. Others

7.1 How to print the message context?

The examples use `ESP_LOG_BUFFER_HEX()` to print the message context while the ESP-BLE-MESH protocol stack uses `bt_hex()`.

7.2 Which API can be used to restart ESP32?

The API *esp_restart()*.

7.3 How to monitor the remaining space of the stack of a task?

The API *vTaskList()* can be used to print the remaining space of the task stack periodically.

7.4 How to change the level of log without changing the menuconfig output level?

The API *esp_log_level_set()* can be used to change the log output level rather than using menuconfig to change it.

ESP-BLE-MESH Terminology

Table 11: Table 1 ESP-BLE-MESH Terminology - Role

Term	Official Definition	Detailed Explanation
Unprovisioned Device	A device that is not a member of a mesh network is known as an unprovisioned device.	Examples: lighting devices, temperature control devices, manufacturing equipments and electric doors, etc.
Node	A node is a provisioned device.	The role of unprovisioned device will change to node after being provisioned to ESP-BLE-MESH network. Nodes (such as lighting devices, temperature control devices, manufacturing equipments, and electric doors) are devices that can send, receive, or relay messages in ESP-BLE-MESH network, and they can optionally support one or more subnets.
Relay Node	A node that supports the Relay feature and has the Relay feature enabled is known as a Relay node.	Relay nodes can receive and resend ESP-BLE-MESH messages, so the messages can be transferred further. Users can decide whether or not to enable forwarding function of nodes according to nodes' status. Messages can be relayed for multiple times, and each relay is considered as a "hop". Messages can hop up to 126 times, which is enough for message transmission in a wide area.
Proxy Node	A node that supports the Proxy feature and has the Proxy feature enabled is known as a Proxy node.	Proxy nodes receive messages from one bearer (it generally includes advertising bearer and GATT bearer) and resend it from another one. The purpose is to connect communication equipments that only support GATT bearer to ESP-BLE-MESH network. Generally, mobile apps need a Proxy node to access Mesh network. Without Proxy nodes, mobile apps cannot communicate with members in Mesh network.
Friend Node	A node that supports the Friend feature, has the Friend feature enabled, and has a friendship with a node that supports the Low Power feature is known as a Friend node.	Friend node, like the backup of Low Power node (LPN), can store messages that are sent to Low Power node and security updates; the stored information will be transferred to Low Power node when Low Power node needs it. Low Power node must establish "friendship" with another node that supports the Friend Feature to reduce duty cycle of its receiver, thus power consumption of Low Power node can be reduced. Low Power node needs to find a Friend node to establish a friendship with it. The process involved is called "friendship establishment". Cooperation between Low Power node and Friend nodes enables Low Power node to schedule the use of the radio, thus Low Power node can receive messages at an appropriate or lower frequency without the need of keeping listening. Low Power node will poll Friend node to see if there is new message.
Low Power Node	A node that supports the Low Power feature and has a friendship with a node that supports the Friend feature is known as a Low Power node.	By polling, Low Power node gets information from Friend node, such as messages, security updates, and etc.
Provisioner	A node that is capable of adding a device to a mesh network.	The device that can provision unprovisioned devices is called a Provisioner. This process usually needs to be implemented through an app that is typically provided by the product manufacturer and can be used on a gateway, a smartphone, tablet or other carriers.

Table 12: Table 2 ESP-BLE-MESH Terminology - Composition

Term	Official Definition	Detailed Explanation
State	A value representing a condition of an element that is exposed by an element of a node.	Each node in a ESP-BLE-MESH network has an independent set of state values that indicate certain states of the device, like brightness, and color of lighting device. Change of state value will lead to change of the physical state of devices. For example, changing the on/off state of a device is actually turning on/off the device.
Model	A model defines the basic functionality of a node.	A node may contain multiple models, and each model defines basic functionalities of nodes, like the states needed by the nodes, the messages controlling the states, and actions resulted from messages handling. The function implementation of the nodes is based on models, which can be divided into SIG Model and Vendor Model, with the former defined by SIG and latter defined by users.
Element	An addressable entity within a device.	A node can contain one or more elements, with each having a unicast address and one or more models, and the models contained by the same element must not be the same.
Composition Data State	The Composition Data state contains information about a node, the elements it includes, and the supported models.	By reading the value of the Composition Data state, users can know basic information of the node, such as the number of elements, and the models in each element. Provisioner gets this message to further provision the device, such as configuring subscription address and publishing address of nodes.

Table 13: Table 3 ESP-BLE-MESH Terminology - Features

Term	Official Definition	Detailed Explanation
Low Power Feature	The ability to operate within a mesh network at significantly reduced receiver duty cycles only in conjunction with a node supporting the Friend feature.	Low Power feature reduces power consumption of nodes. When a Low Power node is searching for a Friend node, and there are multiple Friend nodes nearby, it selects the most suitable Friend node through algorithm.
Friend Feature	The ability to help a node supporting the Low Power feature to operate by storing messages destined for those nodes.	By enabling friend feature, the node can help to store information for Low Power node. The nodes enabled with friend feature may cause more power and memory consumption.
Relay Feature	The ability to receive and retransmit mesh messages over the advertising bearer to enable larger networks.	The relay feature enables ESP-BLE-MESH messages to hop among nodes for multiple times, and the transmission distance can exceed the range of direct radio transmission between two nodes, thereby covering the entire network. When a node is enabled with the relay feature to relay messages, it only relays the messages of its own subnet, and does not relay the messages of other subnets. The data integrity will not be considered when the node enabled with relay feature relays segmented messages. The node would relay every segmented message once it receives one rather than waiting for the complete message.
Proxy Feature	The ability to receive and retransmit mesh messages between GATT and advertising bearers.	The purpose of the proxy feature is to allow nodes without an advertising bearer to access the ESP-BLE-MESH network. The proxy feature is typically used in nodes that need to connect to mobile apps.

Table 14: Table 4 ESP-BLE-MESH Terminology - Provisioning

Term	Official Definition	Detailed Explanation
PB-ADV	PB-ADV is a provisioning bearer used to provision a device using Generic Provisioning PDUs over the advertising channels.	PB-ADV transfers packets generated during the provisioning process over the advertising channels. This way can only be used for provisioning when provisioner and unprovisioned device both support PB-ADV.
PB-GATT	PB-GATT is a provisioning bearer used to provision a device using Proxy PDUs to encapsulate Provisioning PDUs within the Mesh Provisioning Service.	PB-GATT uses connection channels to transfer packets generated during the provisioning process. If an unprovisioned device wants to be provisioned through this method, it needs to implement the related Mesh Provisioning Service. Unprovisioned devices which don ' t implement such service cannot be provisioned into mesh network through PB-GATT bearer.
Provisioning	Provisioning is a process of adding an unprovisioned device to a mesh network, managed by a Provisioner.	The process of provisioning turns the "unprovisioned device" into a "node" , making it a member of the ESP-BLE-MESH network.
Authentication Method	Authentication is a step during the provisioning of nodes.	There are four authentication methods for unprovisioned devices: Output OOB, Input OOB, Static OOB, and No OOB.
Input OOB	Input Out-of-Band	For example, a Provisioner generates and displays a random number, and then prompts users to take appropriate actions to input the random number into the unprovisioned device. Taking lighting switch as an example, users can press the button for several times in a certain period of time to input the random number displayed on the Provisioner. Authentication method of the Input OOB is similar to that of Output OOB, but the role of the device is reversed.
Output OOB	Output Out-of-Band	For example, an unprovisioned device will choose a random number and output the number in a way that is compatible with its functionality. If the unprovisioned device is a bulb, it can flash a specified number of times. If the unprovisioned device has an LCD screen, the random number can display as a multi-digit value. Users who start provisioning should input the observed number to authenticate the unprovisioned device.
Static OOB	Static Out-of-Band	Authentication method of Static OOB: use Static OOB information. Use 0 as Static OOB information if No OOB information is needed. Use Static OOB information to authenticate devices which are going through provisioning if OOB information is needed.
No OOB	No Out-of-Band	Authentication method of No OOB: Set the value of the Static OOB field to 0. Using this way is like not authenticating the unprovisioned devices.

Table 15: Table 5 ESP-BLE-MESH Terminology - Address

Term	Official Definition	Detailed Explanation
Unassigned Address	This is a special address type, with a value of 0x0000. Its use indicates that an Element has not yet been configured or had a Unicast Address assigned to it.	The addresses owned by elements which has not been configured yet or no address has been allocated are unassigned addresses. These elements will not be used for messages transfer because they have no fixed address. Unassigned address is recommended to set as the value of the address before setting the address of user code.
Unicast Address	A unicast address is a unique address allocated to each element.	During provisioning, the Provisioner will assign a unicast address to each element of node within the life cycle of the nodes in the network. A unicast address may appear in the source/destination address field of a message. Messages sent to a unicast address can only be processed by the element that owns the unicast address.
Virtual Address	A virtual address represents a set of destination addresses. Each virtual address logically represents a Label UUID, which is a 128-bit value that does not have to be managed centrally.	Associated with specific UUID labels, a virtual address may serve as the publishing or subscription address of the model. A UUID label is a 128-bit value associated with elements of one or more nodes. For virtual addresses, the 15th and 14th bits are set to 1 and 0 respectively; bits from 13th to 0 are set to hash values (providing 16384 hash values). The hash is a derivation of the Label UUID. To use subscribing elements to check the full 128-bit UUID is very inefficient while hash values provide a more efficient way to determine which elements that which messages are finally sent to.
Group Address	A group address is an address that is programmed into zero or more elements	Group address is another kind of multicast address in the ESP-BLE-MESH network, which is usually used to group nodes. A message sent to the all-proxies address shall be processed by the primary element of all nodes that have the proxy functionality enabled. A message sent to the all-friends address shall be processed by the primary element of all nodes that have the friend functionality enabled. A message sent to the all-relays address shall be processed by the primary element of all nodes that have the relay functionality enabled. A message sent to the all-nodes address shall be processed by the primary element of all nodes.

Table 16: Table 6 ESP-BLE-MESH Terminology - Security

Term	Official Definition	Detailed Explanation
Device Key (DevKey)	There is also a device key, which is a special application key that is unique to each node, is known only to the node and a Configuration Client, and is used to secure communications between the node and a Configuration Client.	The device key enables you to provision the devices, configure the nodes. The device key is used to encrypt Configuration Messages, i.e. the message transferred between the Provisioner and the node when the device is configured.
Application Key (AppKey)	Application keys are used to secure communications at the upper transport layer.	Application key is used for decryption of application data before delivering application data to application layer and encryption of them during the delivery of application layer. Some nodes in the network have a specific purpose and can restrict access to potentially sensitive data based on the needs of the application. With specific application keys, these nodes are associated with specific applications. Generally speaking, the fields using different application keys include security (access control of buildings, machine rooms and CEO offices), lighting (plant, exterior building and sidewalks) and HVAC systems. Application keys are bound to Network keys. This means application keys are only used in a context of a Network key they are bound to. An application key shall only be bound to a single Network key.
Master Security Material	The master security material is derived from the network key (NetKey) and can be used by other nodes in the same network. Messages encrypted with master security material can be decoded by any node in the same network.	The corresponding friendship messages encrypted with the friendship security material: 1. Friend Poll, 2. Friend Update, 3. Friend Subscription List, add/delete/confirm, 4. The Stored Messages" sent by friend nodes to Low Power node. The corresponding friendship messages encrypted with the master security material: 1. Friend Clear, 2. Friend Clear Confirm. Based on the setup of the applications, the messages sent from the Low Power node to the friend nodes will be encrypted with the friendship security material or master security material, with the former being used by the messages transmitted between Low Power node and friend nodes and the latter being used by other network messages.

Table 17: Table 7 ESP-BLE-MESH Terminology - Message

Term	Official Definition	Detailed Explanation
Reassembly / Segmentation	Segmentation and reassembly (SAR) is a method of communication network, which is divided into small units before transmitting packets and reassembled in a proper order at the communication receiving end.	The lower transport layer will automatically segment the message whose size is too big. The receiving end will return a response message, and the transmitting end will send the data packet again that the receiving end does not receive according to the response message. This is automatically completed by the lower transport layer. Unsegmented messages have at most 15 bytes, of which 4 bytes are transMIC, so the remaining is 11 bytes; in the case of segmentation, there are 12 valid bytes in the first several packets, and 8 in the last one. Special case: A shorter packet requires mandatory segmentation from lower transport layer, in which case the valid byte is 8 bytes.
Unacknowledged / Acknowledged	There are two types of messages: Unacknowledged or Acknowledged	Based on the whether or not the receiving end needs to send the response message, the messages sent are divided into two kinds. The sending end should set the maximum number of retransmission.

Table 18: Table 8 ESP-BLE-MESH Terminology - Foundation Models

Term	Official Definition	Detailed Explanation
Configuration Server Model	This model is used to represent a mesh network configuration of a device.	The node must contain the Configuration Server Model, which is responsible for maintaining configuration-related states. The states that Configuration Server Model maintains include: NetKey List, AppKey List, Model to AppKey List, Node Identity, Key Refresh Phase, Heartbeat Publish, Heartbeat Subscription, Network Transmit, Relay Retransmit etc.
Configuration Client Model	The model is used to represent an element that can control and monitor the configuration of a node.	The Configuration Client Model uses messages to control the state maintained by the Configuration Server Model. The Provisioner must contain the Configuration Client Model, with which the configuration messages, like Configuration Composition Data Get can be sent.
Health Server Model	This model is used to represent a mesh network diagnostics of a device.	The Health Server Model is primarily used by devices to check their states and see if there is an error. The states maintained by Health Server model include: Current Fault, Registered Fault, Health Period, and Attention Timer.
Health Client Model	The model is used to represent an element that can control and monitor the health of a node.	The Health Client Model uses messages to control the state maintained by the Health Server Model. The model can get the self-test information of other nodes through the message "Health Fault Get" .

Table 19: Table 9 ESP-BLE-MESH Terminology - Network Management

Term	Official Definition	Detailed Explanation
Key Refresh procedure	This procedure is used when the security of one or more network keys and/or one or more of the application keys has been compromised or could be compromised.	Key Refresh Procedure is used to update network key and application key of ESP-BLE-MESH network. Key Refresh Procedure is used when the security of one or more network keys and/or one or more application keys is threatened or potentially threatened. Keys are usually updated after some nodes in the network are removed.
IV (Initialisation Vector) Update Procedure	A node can also use an IV Update procedure to signal to peer nodes that it is updating the IV Index.	The IV Update procedure is used to update the value of ESP-BLE-MESH network' s IV Index. This value is related to the random number required for message encryption. To ensure that the value of the random number is not repeated, this value is periodically incremented. IV Index is a 32-bit value and a shared network resource. For example, all nodes in a mesh network share the same IV Index value. Starting from 0x00000000, the IV Index increments during the IV Update procedure and maintained by a specific process, ensuring the IV Index shared in the mesh network is the same. This can be done when the node believes that it has the risk of exhausting its sequence number, or when it determines that another node is nearly exhausting its sequence number. Note: The update time must not be less than 96 hours. It can be triggered when a secure network beacon is received, or when the node determines that its sequence number is greater than a certain value.

For more terms, please see: ESP-BLE-MESH Glossary of Terms.

Bluetooth SIG Documentation

- BLE Mesh Core Specification
- BLE Mesh Model Specification
- An Intro to Bluetooth Mesh Part 1 / Part 2

- The Fundamental Concepts of Bluetooth Mesh Networking, Part 1 / Part 2
- Bluetooth Mesh Networking: Friendship
- Management of Devices in a Bluetooth Mesh Network
- Bluetooth Mesh Security Overview
- Provisioning a Bluetooth Mesh Network Part 1 / Part 2

4.10 ESP-WIFI-MESH

This guide provides information regarding the ESP-WIFI-MESH protocol. Please see the *ESP-WIFI-MESH API Reference* for more information about API usage.

4.10.1 Overview

ESP-WIFI-MESH is a networking protocol built atop the Wi-Fi protocol. ESP-WIFI-MESH allows numerous devices (henceforth referred to as nodes) spread over a large physical area (both indoors and outdoors) to be interconnected under a single WLAN (Wireless Local-Area Network). ESP-WIFI-MESH is self-organizing and self-healing meaning the network can be built and maintained autonomously.

The ESP-WIFI-MESH guide is split into the following sections:

1. *Introduction*
2. *ESP-WIFI-MESH Concepts*
3. *Building a Network*
4. *Managing a Network*
5. *Data Transmission*
6. *Channel Switching*
7. *Performance*
8. *Further Notes*

4.10.2 Introduction

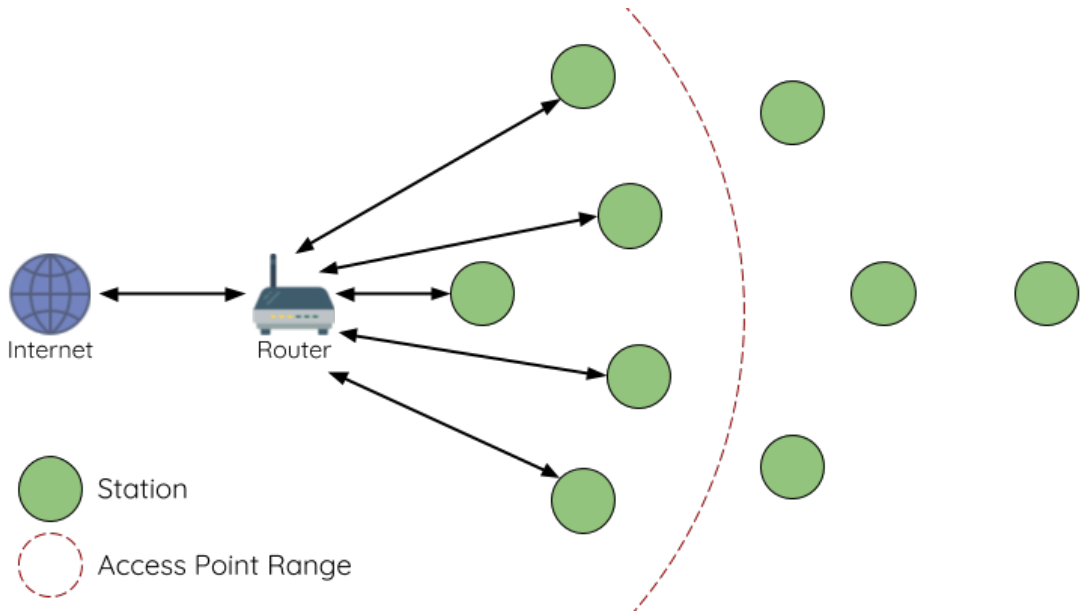


Fig. 19: Traditional Wi-Fi Network Architecture

A traditional infrastructure Wi-Fi network is a point-to-multipoint network where a single central node known as the access point (AP) is directly connected to all other nodes known as stations. The AP is responsible for arbitrating and forwarding transmissions between the stations. Some APs also relay transmissions to/from an external IP network via a router. Traditional infrastructure Wi-Fi networks suffer the disadvantage of limited coverage area due to the requirement that every station must be in range to directly connect with the AP. Furthermore, traditional Wi-Fi networks are susceptible to overloading as the maximum number of stations permitted in the network is limited by the capacity of the AP.

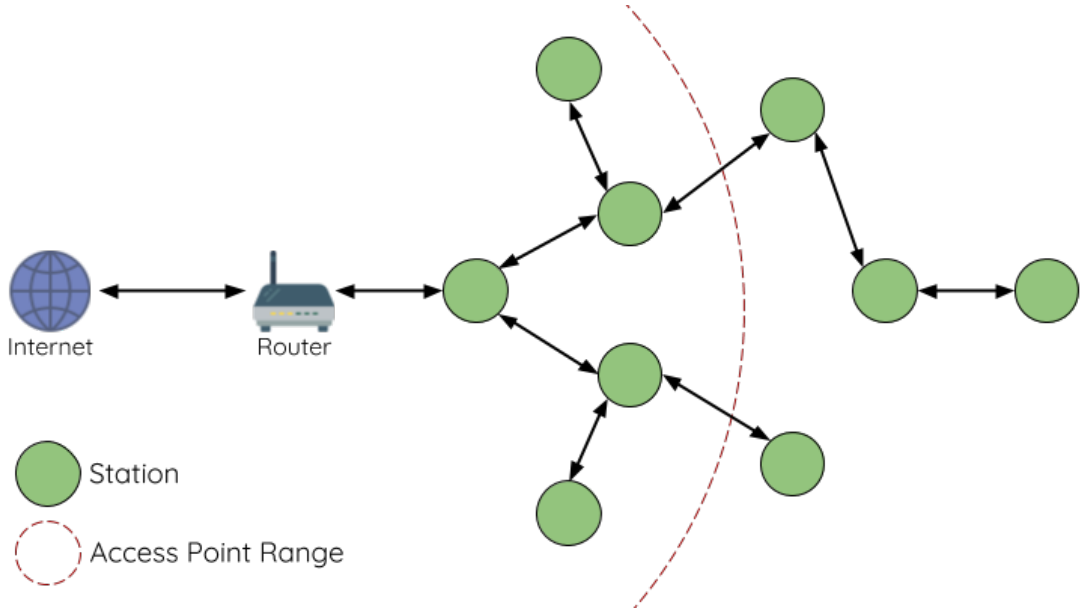


Fig. 20: ESP-WIFI-MESH Network Architecture

ESP-WIFI-MESH differs from traditional infrastructure Wi-Fi networks in that nodes are not required to connect to a central node. Instead, nodes are permitted to connect with neighboring nodes. Nodes are mutually responsible for relaying each others transmissions. This allows an ESP-WIFI-MESH network to have much greater coverage area as nodes can still achieve interconnectivity without needing to be in range of the central node. Likewise, ESP-WIFI-MESH is also less susceptible to overloading as the number of nodes permitted on the network is no longer limited by a single central node.

4.10.3 ESP-WIFI-MESH Concepts

Terminology

Term	Description
Node	Any device that **is** or **can be** part of an ESP-WIFI-MESH network
Root Node	The top node in the network
Child Node	A node X is a child node when it is connected to another node Y where the connection makes node X more distant from the root node than node Y (in terms of number of connections).
Parent Node	The converse notion of a child node
Descendant Node	Any node reachable by repeated proceeding from parent to child
Sibling Nodes	Nodes that share the same parent node
Connection	A traditional Wi-Fi association between an AP and a station. A node in ESP-WIFI-MESH will use its station interface to associate with the softAP interface of another node, thus forming a connection. The connection process includes the authentication and association processes in Wi-Fi.
Upstream Connection	The connection from a node to its parent node
Downstream Connection	The connection from a node to one of its child nodes
Wireless Hop	The portion of the path between source and destination nodes that corresponds to a single wireless connection. A data packet that traverses a single connection is known as **single-hop** whereas traversing multiple connections is known as **multi-hop**.
Subnetwork	A subnetwork is subdivision of an ESP-WIFI-MESH network which consists of a node and all of its descendant nodes. Therefore the subnetwork of the root node consists of all nodes in an ESP-WIFI-MESH network.
MAC Address	Media Access Control Address used to uniquely identify each node or router within an ESP-WIFI-MESH network.
DS	Distribution System (External IP Network)

Tree Topology

ESP-WIFI-MESH is built atop the infrastructure Wi-Fi protocol and can be thought of as a networking protocol that combines many individual Wi-Fi networks into a single WLAN. In Wi-Fi, stations are limited to a single connection with an AP (upstream connection) at any time, whilst an AP can be simultaneously connected to multiple stations (downstream connections). However ESP-WIFI-MESH allows nodes to simultaneously act as a station and an AP. Therefore a node in ESP-WIFI-MESH can have **multiple downstream connections using its softAP interface**, whilst simultaneously having **a single upstream connection using its station interface**. This naturally results in a tree network topology with a parent-child hierarchy consisting of multiple layers.

ESP-WIFI-MESH is a multiple hop (multi-hop) network meaning nodes can transmit packets to other nodes in the network through one or more wireless hops. Therefore, nodes in ESP-WIFI-MESH not only transmit their own packets, but simultaneously serve as relays for other nodes. Provided that a path exists between any two nodes on the physical layer (via one or more wireless hops), any pair of nodes within an ESP-WIFI-MESH network can communicate.

Note: The size (total number of nodes) in an ESP-WIFI-MESH network is dependent on the maximum number of layers permitted in the network, and the maximum number of downstream connections each node can have. Both of these variables can be configured to limit the size of the network.

Node Types

Root Node: The root node is the top node in the network and serves as the only interface between the ESP-WIFI-MESH network and an external IP network. The root node is connected to a conventional Wi-Fi router and relays packets to/from the external IP network to nodes within the ESP-WIFI-MESH network. **There can only be one root node within an ESP-WIFI-MESH network** and the root node's upstream connection may only be with the router. Referring to the diagram above, node A is the root node of the network.

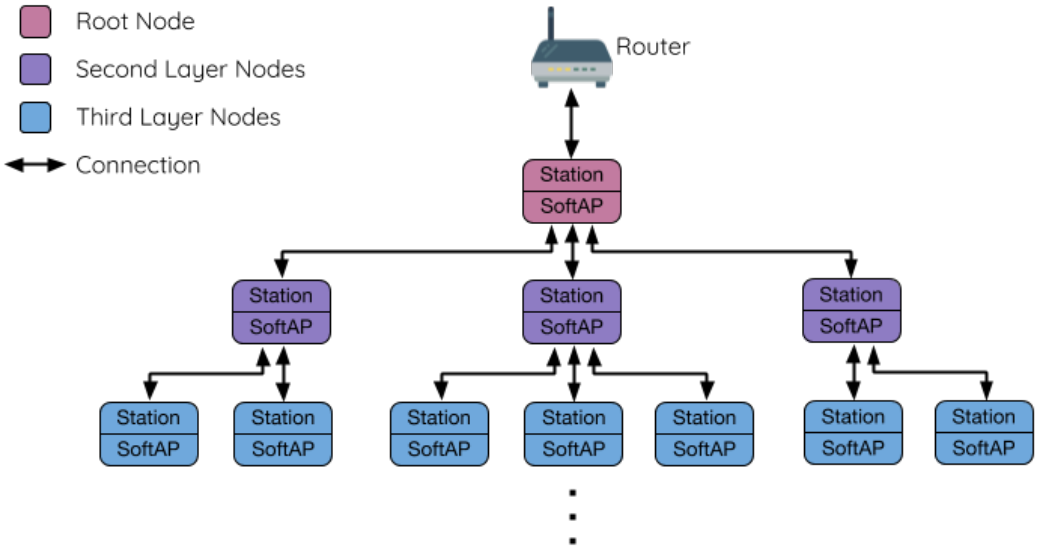


Fig. 21: ESP-WIFI-MESH Tree Topology

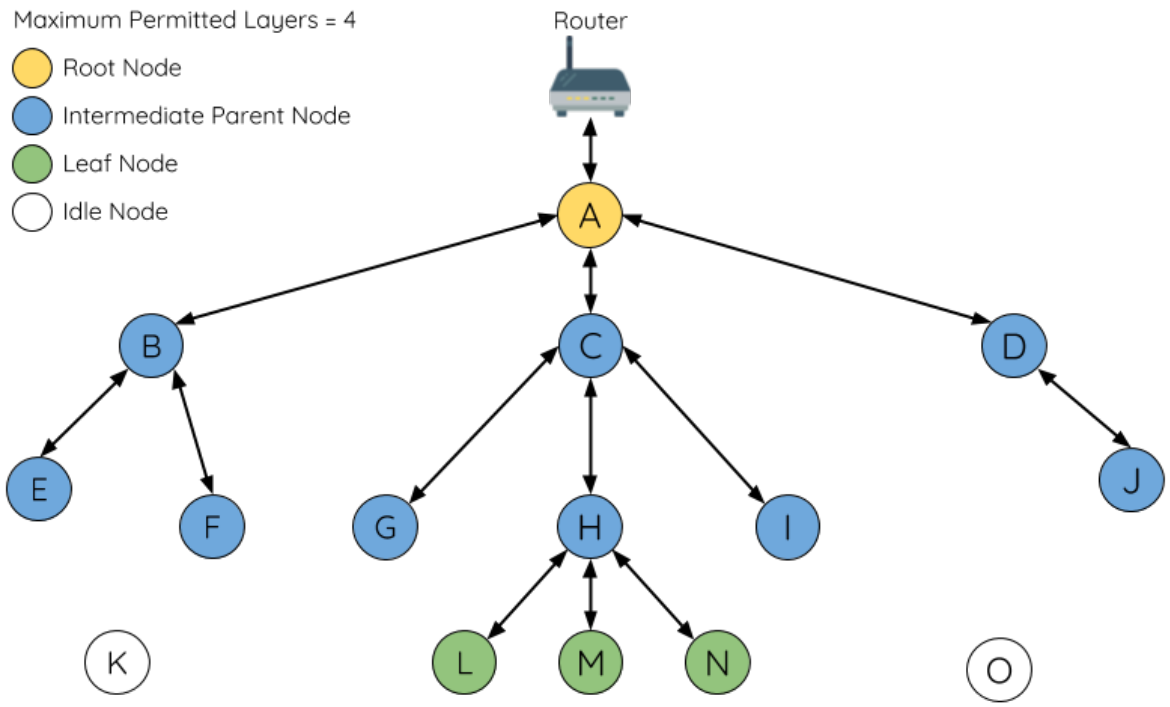


Fig. 22: ESP-WIFI-MESH Node Types

Leaf Nodes: A leaf node is a node that is not permitted to have any child nodes (no downstream connections). Therefore a leaf node can only transmit or receive its own packets, but cannot forward the packets of other nodes. If a node is situated on the network ' s maximum permitted layer, it will be assigned as a leaf node. This prevents the node from forming any downstream connections thus ensuring the network does not add an extra layer. Some nodes without a softAP interface (station only) will also be assigned as leaf nodes due to the requirement of a softAP interface for any downstream connections. Referring to the diagram above, nodes L/M/N are situated on the networks maximum permitted layer hence have been assigned as leaf nodes .

Intermediate Parent Nodes: Connected nodes that are neither the root node or a leaf node are intermediate parent nodes. An intermediate parent node must have a single upstream connection (a single parent node), but can have zero to multiple downstream connections (zero to multiple child nodes). Therefore an intermediate parent node can transmit and receive packets, but also forward packets sent from its upstream and downstream connections. Referring to the diagram above, nodes B to J are intermediate parent nodes. **Intermediate parent nodes without downstream connections such as nodes E/F/G/I/J are not equivalent to leaf nodes** as they are still permitted to form downstream connections in the future.

Idle Nodes: Nodes that have yet to join the network are assigned as idle nodes. Idle nodes will attempt to form an upstream connection with an intermediate parent node or attempt to become the root node under the correct circumstances (see *Automatic Root Node Selection*). Referring to the diagram above, nodes K and O are idle nodes.

Beacon Frames & RSSI Thresholding

Every node in ESP-WIFI-MESH that is able to form downstream connections (i.e. has a softAP interface) will periodically transmit Wi-Fi beacon frames. A node uses beacon frames to allow other nodes to detect its presence and know of its status. Idle nodes will listen for beacon frames to generate a list of potential parent nodes, one of which the idle node will form an upstream connection with. ESP-WIFI-MESH uses the Vendor Information Element to store metadata such as:

- Node Type (Root, Intermediate Parent, Leaf, Idle)
- Current layer of Node
- Maximum number of layers permitted in the network
- Current number of child nodes
- Maximum number of downstream connections to accept

The signal strength of a potential upstream connection is represented by RSSI (Received Signal Strength Indication) of the beacon frames of the potential parent node. To prevent nodes from forming a weak upstream connection, ESP-WIFI-MESH implements an RSSI threshold mechanism for beacon frames. If a node detects a beacon frame with an RSSI below a preconfigured threshold, the transmitting node will be disregarded when forming an upstream connection.

Panel A of the illustration above demonstrates how the RSSI threshold affects the number of parent node candidates an idle node has.

Panel B of the illustration above demonstrates how an RF shielding object can lower the RSSI of a potential parent node. Due to the RF shielding object, the area in which the RSSI of node X is above the threshold is significantly reduced. This causes the idle node to disregard node X even though node X is physically adjacent. The idle node will instead form an upstream connection with the physically distant node Y due to a stronger RSSI.

Note: Nodes technically still receive all beacon frames on the MAC layer. The RSSI threshold is an ESP-WIFI-MESH feature that simply filters out all received beacon frames that are below the preconfigured threshold.

Preferred Parent Node

When an idle node has multiple parent nodes candidates (potential parent nodes), the idle node will form an upstream connection with the **preferred parent node**. The preferred parent node is determined based on the following criteria:

- Which layer the parent node candidate is situated on
- The number of downstream connections (child nodes) the parent node candidate currently has

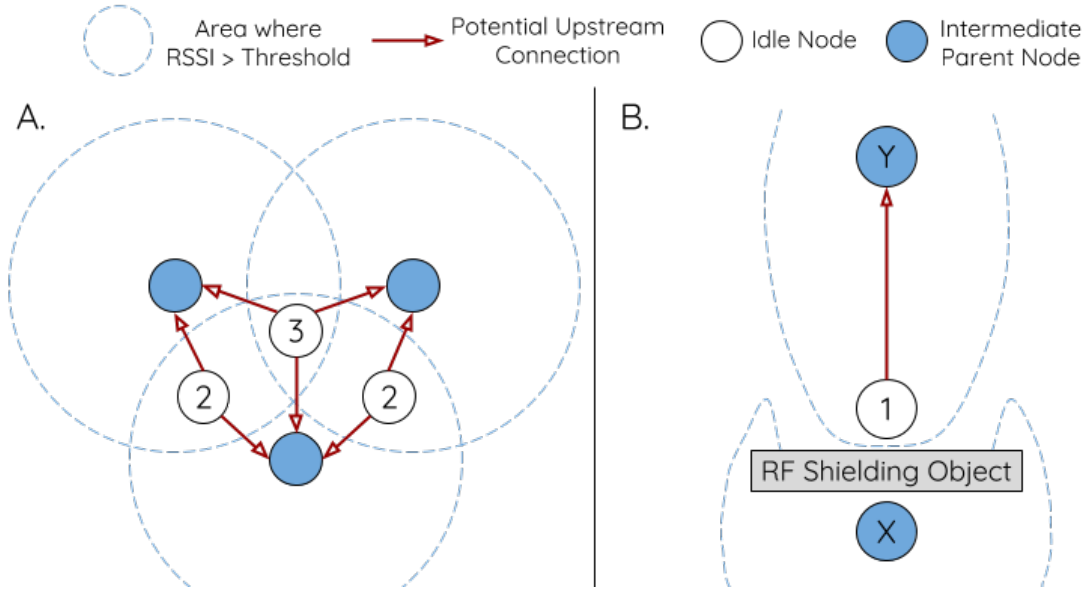


Fig. 23: Effects of RSSI Thresholding

The selection of the preferred parent node will always prioritize the parent node candidate on the shallowest layer of the network (including the root node). This helps minimize the total number of layers in an ESP-WIFI-MESH network when upstream connections are formed. For example, given a second layer node and a third layer node, the second layer node will always be preferred.

If there are multiple parent node candidates within the same layer, the parent node candidate with the least child nodes will be preferred. This criteria has the effect of balancing the number of downstream connections amongst nodes of the same layer.

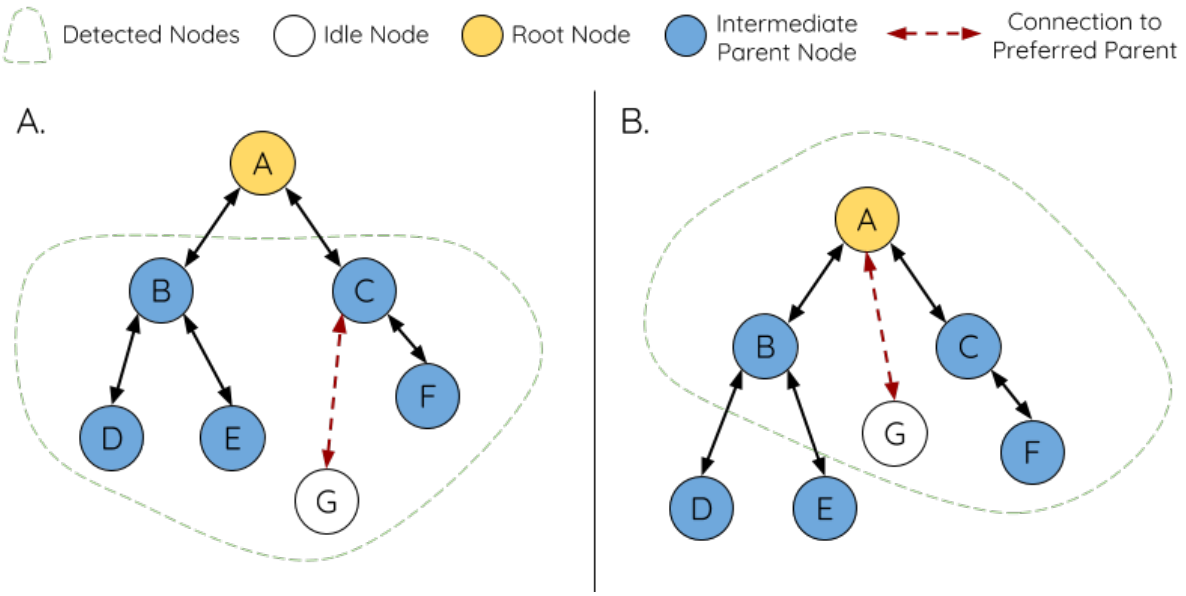


Fig. 24: Preferred Parent Node Selection

Panel A of the illustration above demonstrates an example of how the idle node G selects a preferred parent node given the five parent node candidates B/C/D/E/F. Nodes on the shallowest layer are preferred, hence nodes B/C are prioritized since they are second layer nodes whereas nodes D/E/F are on the third layer. Node C is selected as the preferred parent node due it having fewer downstream connections (fewer child nodes) compared to node B.

Panel B of the illustration above demonstrates the case where the root node is within range of the idle node G. In

other words, the root node's beacon frames are above the RSSI threshold when received by node G. The root node is always the shallowest node in an ESP-WIFI-MESH network hence is always the preferred parent node given multiple parent node candidates.

Note: Users may also define their own algorithm for selecting a preferred parent node, or force a node to only connect with a specific parent node (see the Mesh Manual Networking Example).

Routing Tables

Each node within an ESP-WIFI-MESH network will maintain its individual routing table used to correctly route ESP-WIFI-MESH packets (see *ESP-WIFI-MESH Packet*) to the correct destination node. The routing table of a particular node will **consist of the MAC addresses of all nodes within the particular node's subnetwork** (including the MAC address of the particular node itself). Each routing table is internally partitioned into multiple subtables with each subtable corresponding to the subnetwork of each child node.

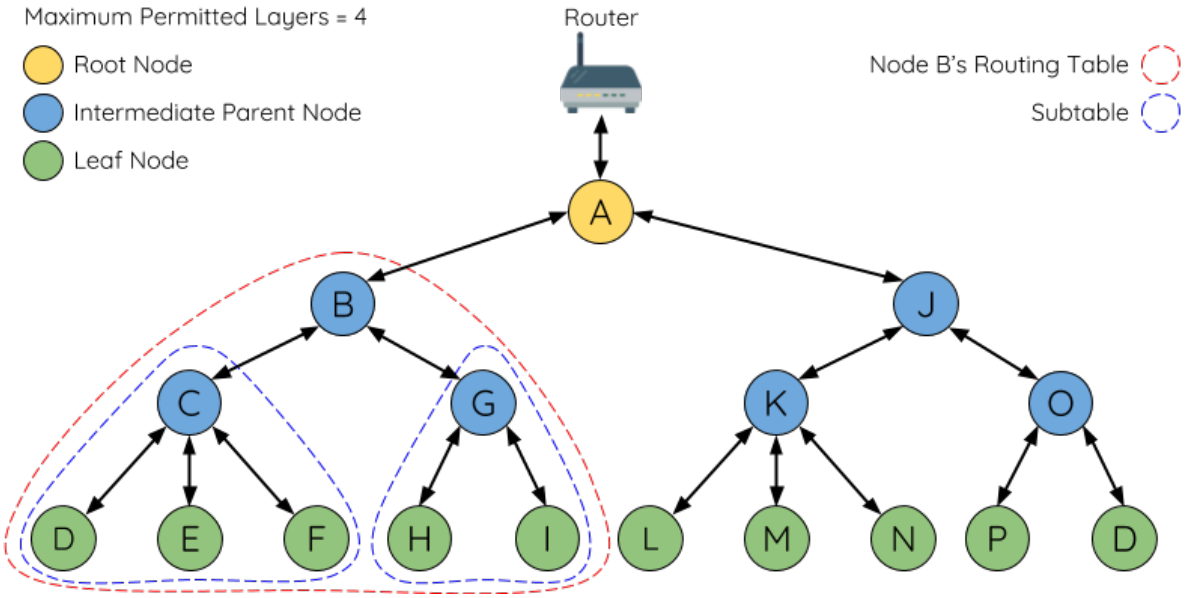


Fig. 25: ESP-WIFI-MESH Routing Tables Example

Using the diagram above as an example, the routing table of node B would consist of the MAC addresses of nodes B to I (i.e. equivalent to the subnetwork of node B). Node B's routing table is internally partitioned into two subtables containing of nodes C to F and nodes G to I (i.e. equivalent to the subnetworks of nodes C and G respectively).

ESP-WIFI-MESH utilizes routing tables to determine whether an ESP-WIFI-MESH packet should be forwarded upstream or downstream based on the following rules.

1. If the packet's destination MAC address is within the current node's routing table and is not the current node, select the subtable that contains the destination MAC address and forward the data packet downstream to the child node corresponding to the subtable.

2. If the destination MAC address is not within the current node's routing table, forward the data packet upstream to the current node's parent node. Doing so repeatedly will result in the packet arriving at the root node where the routing table should contain all nodes within the network.

Note: Users can call `esp_mesh_get_routing_table()` to obtain a node's routing table, or `esp_mesh_get_routing_table_size()` to obtain the size of a node's routing table. `esp_mesh_get_subnet_nodes_list()` can be used to obtain the corresponding subtable of a specific child node. Likewise `esp_mesh_get_subnet_nodes_num()` can be used to obtain the size of the

subtable.

4.10.4 Building a Network

General Process

Warning: Before the ESP-WIFI-MESH network building process can begin, certain parts of the configuration must be uniform across each node in the network (see `mesh_cfg_t`). Each node must be configured with **the same Mesh Network ID, router configuration, and softAP configuration**.

An ESP-WIFI-MESH network building process involves selecting a root node, then forming downstream connections layer by layer until all nodes have joined the network. The exact layout of the network can be dependent on factors such as root node selection, parent node selection, and asynchronous power-on reset. However, the ESP-WIFI-MESH network building process can be generalized into the following steps:

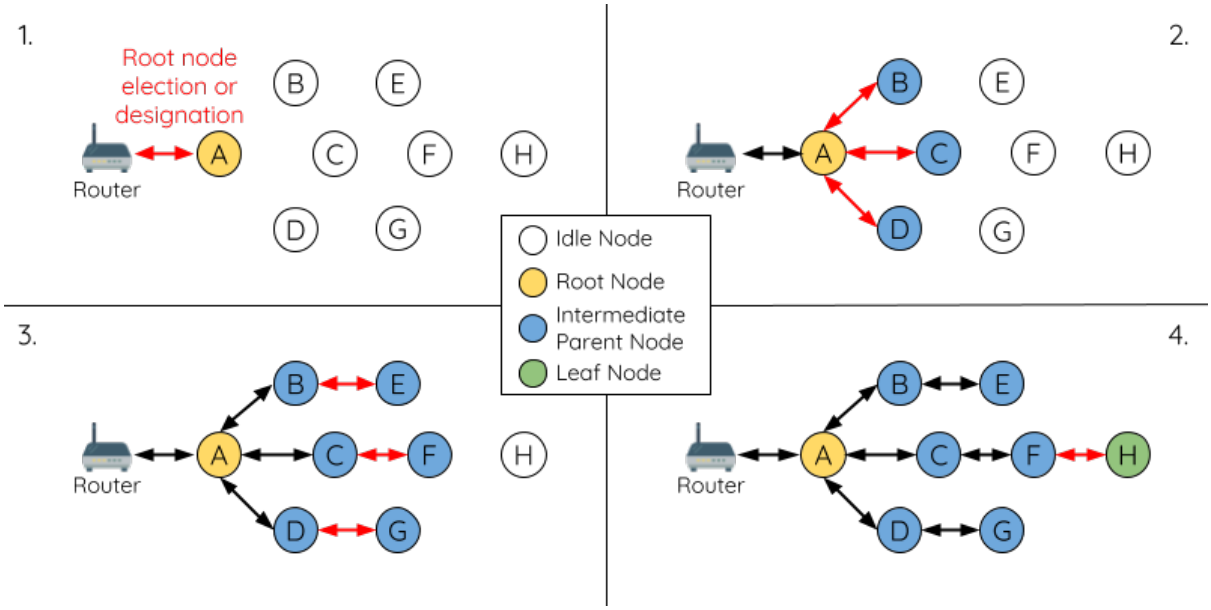


Fig. 26: ESP-WIFI-MESH Network Building Process

1. Root Node Selection The root node can be designated during configuration (see section on *User Designated Root Node*), or dynamically elected based on the signal strength between each node and the router (see *Automatic Root Node Selection*). Once selected, the root node will connect with the router and begin allowing downstream connections to form. Referring to the figure above, node A is selected to be the root node hence node A forms an upstream connection with the router.

2. Second Layer Formation Once the root node has connected to the router, idle nodes in range of the root node will begin connecting with the root node thereby forming the second layer of the network. Once connected, the second layer nodes become intermediate parent nodes (assuming maximum permitted layers > 2) hence the next layer to form. Referring to the figure above, nodes B to D are in range of the root node. Therefore nodes B to D form upstream connections with the root node and become intermediate parent nodes.

3. Formation of remaining layers The remaining idle nodes will connect with intermediate parent nodes within range thereby forming a new layer in the network. Once connected, the idles nodes become intermediate parent node or leaf nodes depending on the networks maximum permitted layers. This step is repeated until there are no more idle

nodes within the network or until the maximum permitted layer of the network has been reached. Referring to the figure above, nodes E/F/G connect with nodes B/C/D respectively and become intermediate parent nodes themselves.

4. Limiting Tree Depth To prevent the network from exceeding the maximum permitted number of layers, nodes on the maximum layer will automatically become leaf nodes once connected. This prevents any other idle node from connecting with the leaf node thereby prevent a new layer form forming. However if an idle node has no other potential parent node, it will remain idle indefinitely. Referring to the figure above, the network's number of maximum permitted layers is set to four. Therefore when node H connects, it becomes a leaf node to prevent any downstream connections from forming.

Automatic Root Node Selection

The automatic selection of a root node involves an election process amongst all idle nodes based on their signal strengths with the router. Each idle node will transmit their MAC addresses and router RSSI values via Wi-Fi beacon frames. **The MAC address is used to uniquely identify each node in the network** whilst the **router RSSI** is used to indicate a node's signal strength with reference to the router.

Each node will then simultaneously scan for the beacon frames from other idle nodes. If a node detects a beacon frame with a stronger router RSSI, the node will begin transmitting the contents of that beacon frame (i.e. voting for the node with the stronger router RSSI). The process of transmission and scanning will repeat for a preconfigured minimum number of iterations (10 iterations by default) and result in the beacon frame with the strongest router RSSI being propagated throughout the network.

After all iterations, each node will individually check for its **vote percentage** (`number of votes/number of nodes participating in election`) to determine if it should become the root node. **If a node has a vote percentage larger than a preconfigured threshold (90% by default), the node will become a root node**.

The following diagram demonstrates how an ESP-WIFI-MESH network is built when the root node is automatically selected.

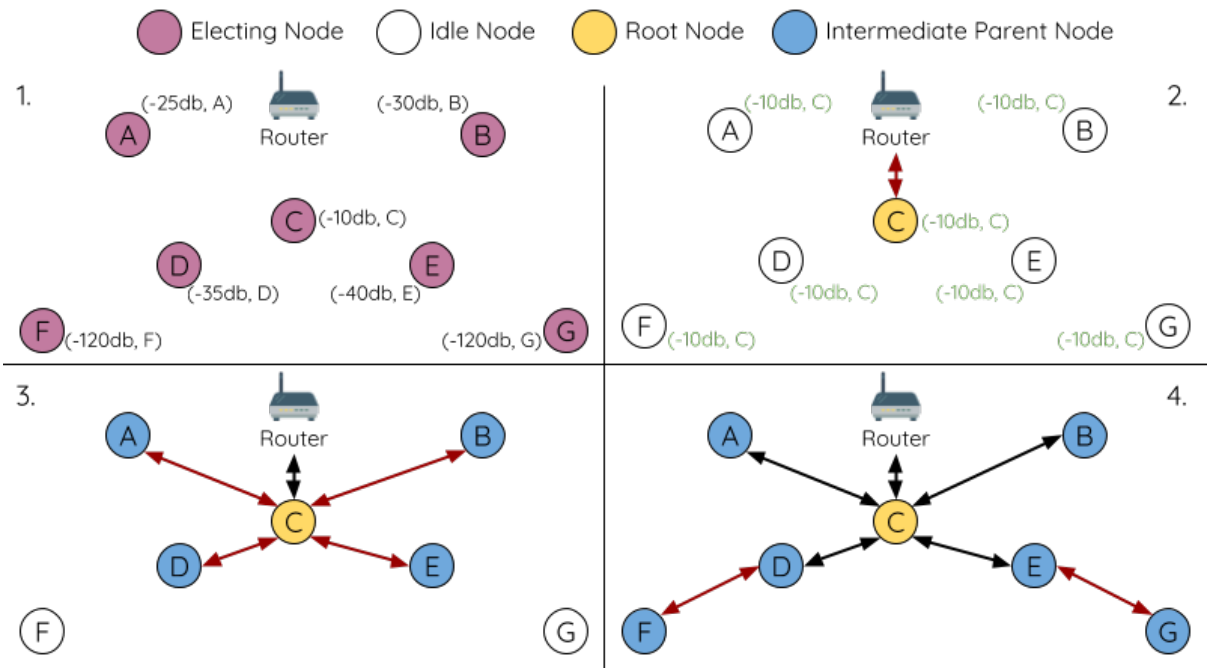


Fig. 27: Root Node Election Example

1. On power-on reset, each node begins transmitting beacon frames consisting of their own MAC addresses and their router RSSIs.

2. Over multiple iterations of transmission and scanning, the beacon frame with the strongest router RSSI is propagated throughout the network. Node C has the strongest router RSSI (-10 dB) hence its beacon frame is propagated

throughout the network. All nodes participating in the election vote for node C thus giving node C a vote percentage of 100%. Therefore node C becomes a root node and connects with the router.

3. Once Node C has connected with the router, nodes A/B/D/E connectwith node C as it is the preferred parent node (i.e. the shallowest node). Nodes A/B/D/E form the second layer of the network.

4. Node F and G connect with nodes D and E respectively and the network building process is complete.

Note: The minimum number of iterations for the election process can be configured using `esp_mesh_set_attempts()`. Users should adjust the number of iterations based on the number of nodes within the network (i.e. the larger the network the larger number of scan iterations required).

Warning: **Vote percentage threshold** can also be configured using *`esp_mesh_set_vote_percentage()`*. Setting a low vote percentage threshold **can result in two or more nodes becoming root nodes** within the same ESP-WIFI-MESH network leading to the building of multiple networks. If such is the case, ESP-WIFI-MESH has internal mechanisms to autonomously resolve the **root node conflict**. The networks of the multiple root nodes will be combined into a single network with a single root node. However, root node conflicts where two or more root nodes have the same router SSID but different router BSSID are not handled.

User Designated Root Node

The root node can also be designated by user which will entail the designated root node to directly connect with the router and forgo the election process. When a root node is designated, all other nodes within the network must also forgo the election process to prevent the occurrence of a root node conflict. The following diagram demonstrates how an ESP-WIFI-MESH network is built when the root node is designated by the user.

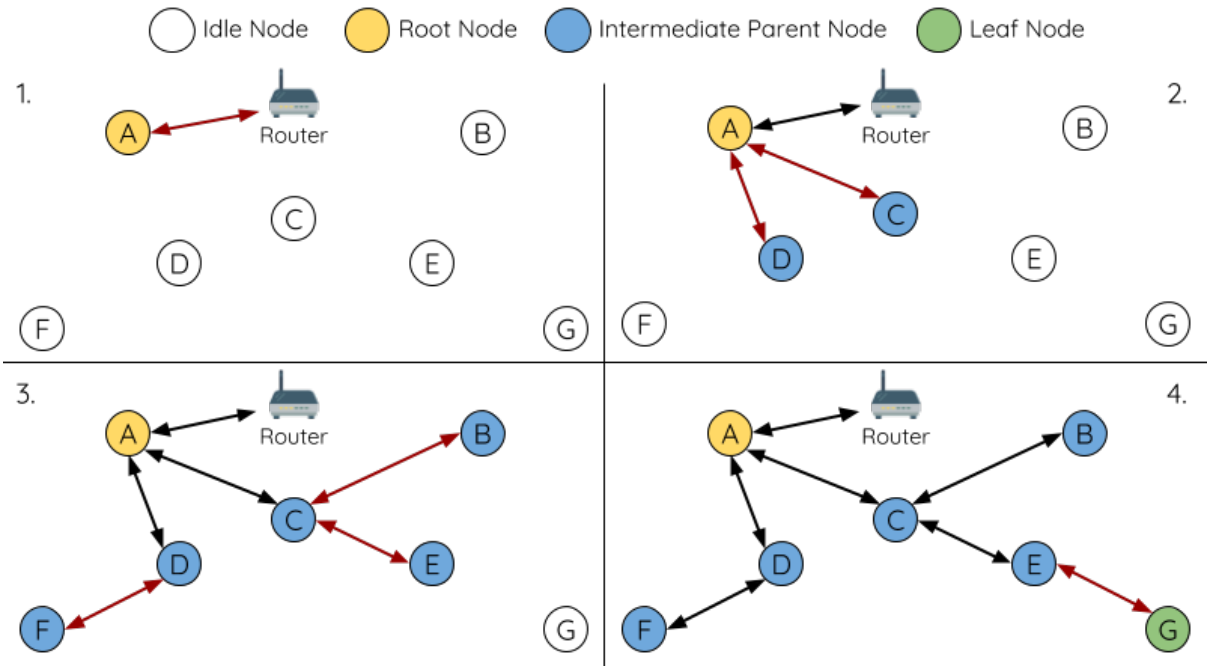


Fig. 28: Root Node Designation Example (Root Node = A, Max Layers = 4)

1. Node A is designated the root node by the user therefore directly connects with the router. All other nodes forgo the election process.

2. Nodes C/D connect with node A as their preferred parent node. Both nodes form the second layer of the network.

3. Likewise, nodes B/E connect with node C, and node F connects with node D. Nodes B/E/F form the third layer of the network.

4. Node G connects with node E, forming the fourth layer of the network. However the maximum permitted number of layers in this network is configured as four, therefore node G becomes a leaf node to prevent any new layers from forming.

Note: When designating a root node, the root node should call `esp_mesh_set_parent()` in order to directly connect with the router. Likewise, all other nodes should call `esp_mesh_fix_root()` to forgo the election process.

Parent Node Selection

By default, ESP-WIFI-MESH is self organizing meaning that each node will autonomously select which potential parent node to form an upstream connection with. The autonomously selected parent node is known as the preferred parent node. The criteria used for selecting the preferred parent node is designed to reduce the number of layers in the ESP-WIFI-MESH network and to balance the number of downstream connections between potential parent nodes (see section on *Preferred Parent Node*).

However ESP-WIFI-MESH also allows users to disable self-organizing behavior which will allow users to define their own criteria for parent node selection, or to configure nodes to have designated parent nodes (see the Mesh Manual Networking Example).

Asynchronous Power-on Reset

ESP-WIFI-MESH network building can be affected by the order in which nodes power-on. If certain nodes within the network power-on asynchronously (i.e. separated by several minutes), **the final structure of the network could differ from the ideal case where all nodes are powered on synchronously**. Nodes that are delayed in powering on will adhere to the following rules:

Rule 1: If a root node already exists in the network, the delayed node will not attempt to elect a new root node, even if it has a stronger RSSI with the router. The delayed node will instead join the network like any other idle node by connecting with a preferred parent node. If the delayed node is the designated root node, all other nodes in the network will remain idle until the delayed node powers-on.

Rule 2: If a delayed node forms an upstream connection and becomes an intermediate parent node, it may also become the new preferred parent of other nodes (i.e. being a shallower node). This will cause the other nodes to switch their upstream connections to connect with the delayed node (see *Parent Node Switching*).

Rule 3: If an idle node has a designated parent node which is delayed in powering-on, the idle node will not attempt to form any upstream connections in the absence of its designated parent node. The idle node will remain idle indefinitely until its designated parent node powers-on.

The following example demonstrates the effects of asynchronous power-on with regards to network building.

1. Nodes A/C/D/F/G/H are powered-on synchronously and begin the root node election process by broadcasting their MAC addresses and router RSSIs. Node A is elected as the root node as it has the strongest RSSI.

2. Once node A becomes the root node, the remaining nodes begin forming upstream connections layer by layer with their preferred parent nodes. The result is a network with five layers.

3. Node B/E are delayed in powering-on but neither attempt to become the root node even though they have stronger router RSSIs (-20 dB and -10 dB) compared to node A. Instead both delayed nodes form upstream connections with their preferred parent nodes A and C respectively. Both nodes B/E become intermediate parent nodes after connecting.

4. Nodes D/G switch their upstream connections as node B is the new preferred parent node due to it being on a shallower layer (second layer node). Due to the switch, the resultant network has three layers instead of the original five layers.

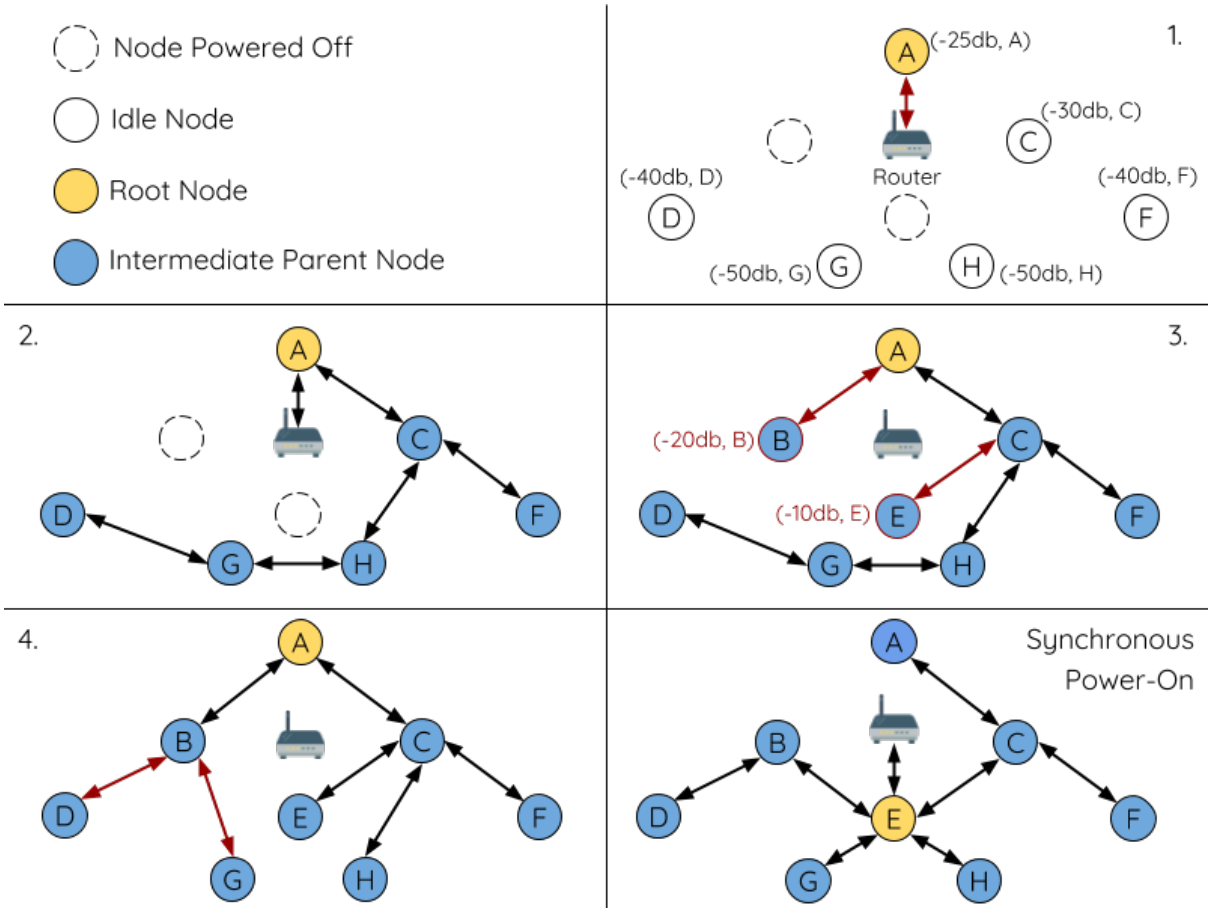


Fig. 29: Network Building with Asynchronous Power On Example

Synchronous Power-On: Had all nodes powered-on synchronously, node E would have become the root node as it has the strongest router RSSI (-10 dB). This would result in a significantly different network layout compared to the network formed under the conditions of asynchronous power-on. **However the synchronous power-on network layout can still be reached if the user manually switches the root node** (see `esp_mesh_waive_root()`).

Note: Differences in parent node selection caused by asynchronous power-on are autonomously corrected for to some extent in ESP-WIFI-MESH (see *Parent Node Switching*)

Loop-back Avoidance, Detection, and Handling

A loop-back is the situation where a particular node forms an upstream connection with one of its descendant nodes (a node within the particular node's subnetwork). This results in a circular connection path thereby breaking the tree topology. ESP-WIFI-MESH prevents loop-back during parent selection by excluding nodes already present in the selecting node's routing table (see *Routing Tables*) thus prevents a particular node from attempting to connect to any node within its subnetwork.

In the event that a loop-back occurs, ESP-WIFI-MESH utilizes a path verification mechanism and energy transfer mechanism to detect the loop-back occurrence. The parent node of the upstream connection that caused the loop-back will then inform the child node of the loop-back and initiate a disconnection.

4.10.5 Managing a Network

ESP-WIFI-MESH is a self healing network meaning it can detect and correct for failures in network routing. Failures occur when a parent node with one or more child nodes breaks down, or when the connection between a parent node and its child nodes becomes unstable. Child nodes in ESP-WIFI-MESH will autonomously select a new parent node and form an upstream connection with it to maintain network interconnectivity. ESP-WIFI-MESH can handle both Root Node Failures and Intermediate Parent Node Failures.

Root Node Failure

If the root node breaks down, the nodes connected with it (second layer nodes) will promptly detect the failure of the root node. The second layer nodes will initially attempt to reconnect with the root node. However after multiple failed attempts, the second layer nodes will initialize a new round of root node election. **The second layer node with the strongest router RSSI will be elected as the new root node** whilst the remaining second layer nodes will form an upstream connection with the new root node (or a neighboring parent node if not in range).

If the root node and multiple downstream layers simultaneously break down (e.g. root node, second layer, and third layer), the shallowest layer that is still functioning will initialize the root node election. The following example illustrates an example of self healing from a root node break down.

1. Node C is the root node of the network. Nodes A/B/D/E are second layer nodes connected to node C.

2. Node C breaks down. After multiple failed attempts to reconnect, the second layer nodes begin the election process by broadcasting their router RSSIs. Node B has the strongest router RSSI.

3. Node B is elected as the root node and begins accepting downstream connections. The remaining second layer nodes A/D/E form upstream connections with node B thus the network is healed and can continue operating normally.

Note: If a designated root node breaks down, the remaining nodes **will not autonomously attempt to elect a new root node** as an election process will never be attempted whilst a designated root node is used.

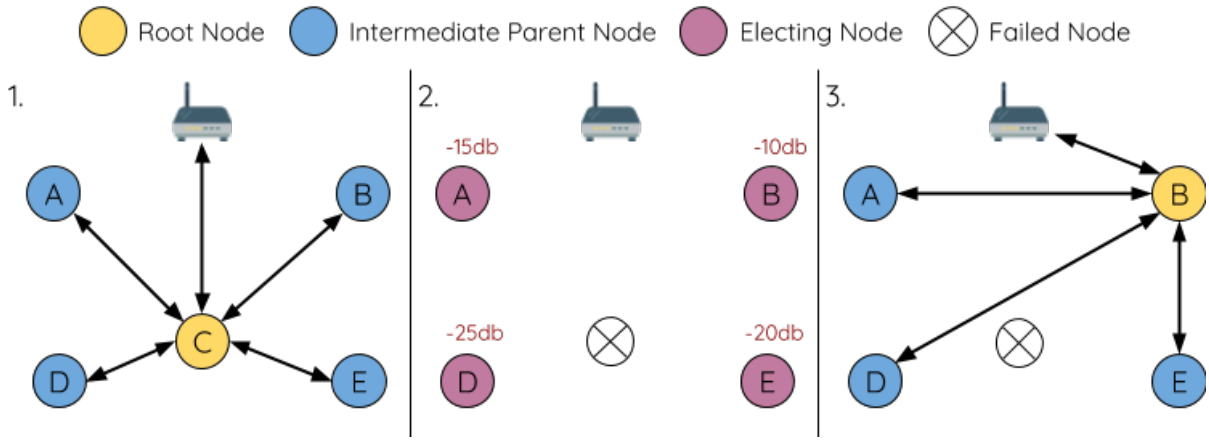


Fig. 30: Self Healing From Root Node Failure

Intermediate Parent Node Failure

If an intermediate parent node breaks down, the disconnected child nodes will initially attempt to reconnect with the parent node. After multiple failed attempts to reconnect, each child node will begin to scan for potential parent nodes (see *Beacon Frames & RSSI Thresholding*).

If other potential parent nodes are available, each child node will individually select a new preferred parent node (see *Preferred Parent Node*) and form an upstream connection with it. If there are no other potential parent nodes for a particular child node, it will remain idle indefinitely.

The following diagram illustrates an example of self healing from an Intermediate Parent Node break down.

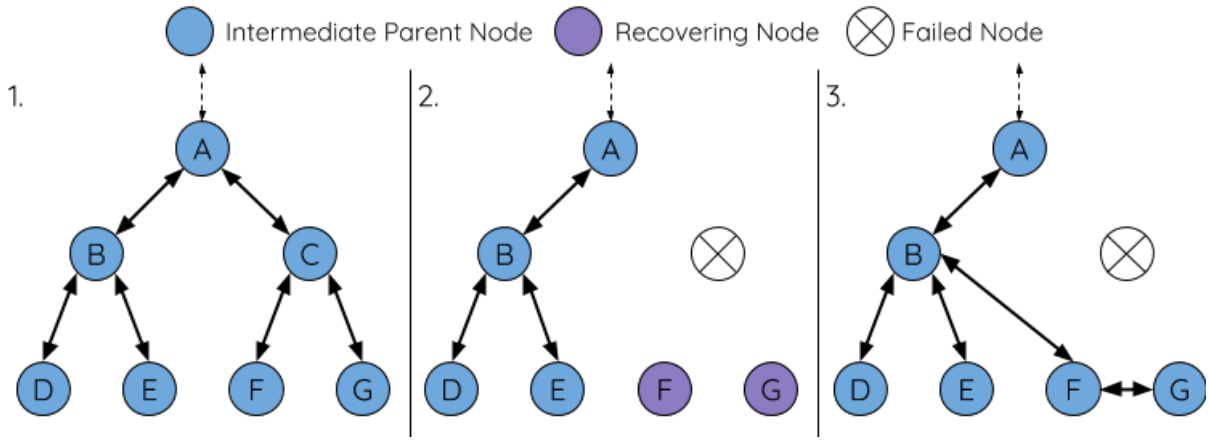


Fig. 31: Self Healing From Intermediate Parent Node Failure

1. The following branch of the network consists of nodes A to G.

2. Node C breaks down. Nodes F/G detect the break down and attempt to reconnect with node C. After multiple failed attempts to reconnect, nodes F/G begin to select a new preferred parent node.

3. Node G is out of range from any other parent node hence remains idle for the time being. Node F is in range of nodes B/E, however node B is selected as it is the shallower node. Node F becomes an intermediate parent node after connecting with Node B thus node G can connect with node F. The network is healed, however the network routing as been affected and an extra layer has been added.

Note: If a child node has a designated parent node that breaks down, the child node will make no attempt to connect with a new parent node. The child node will remain idle indefinitely.

Root Node Switching

ESP-WIFI-MESH does not automatically switch the root node unless the root node breaks down. Even if the root node's router RSSI degrades to the point of disconnection, the root node will remain unchanged. Root node switching is the act of explicitly starting a new election such that a node with a stronger router RSSI will be elected as the new root node. This can be a useful method of adapting to degrading root node performance.

To trigger a root node switch, the current root node must explicitly call `esp_mesh_waive_root()` to trigger a new election. The current root node will signal all nodes within the network to begin transmitting and scanning for beacon frames (see *Automatic Root Node Selection*) **whilst remaining connected to the network (i.e. not idle)**. If another node receives more votes than the current root node, a root node switch will be initiated. **The root node will remain unchanged otherwise**.

A newly elected root node sends a **switch request** to the current root node which in turn will respond with an acknowledgment signifying both nodes are ready to switch. Once the acknowledgment is received, the newly elected root node will disconnect from its parent and promptly form an upstream connection with the router thereby becoming the new root node of the network. The previous root node will disconnect from the router **whilst maintaining all of its downstream connections** and enter the idle state. The previous root node will then begin scanning for potential parent nodes and selecting a preferred parent.

The following diagram illustrates an example of a root node switch.

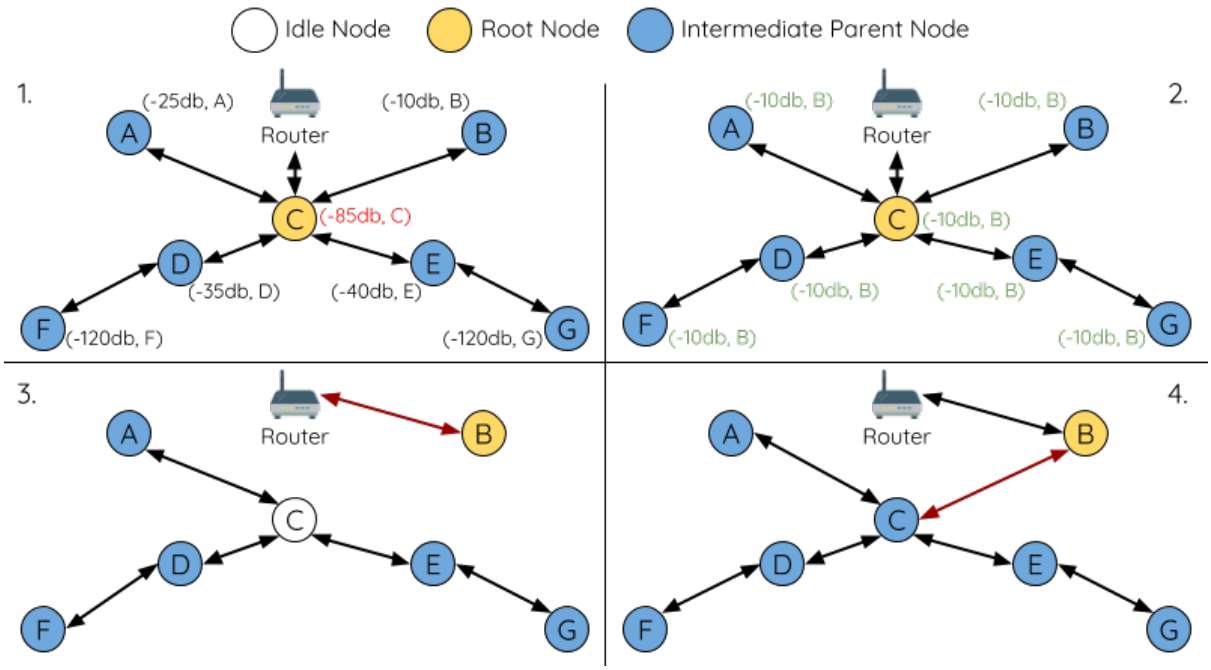


Fig. 32: Root Node Switch Example

1. Node C is the current root node but has degraded signal strength with the router (-85db). The node C triggers a new election and all nodes begin transmitting and scanning for beacon frames **whilst still being connected**.

2. After multiple rounds of transmission and scanning, node B is elected as the new root node. Node B sends node C a **switch request** and node C responds with an acknowledgment.

3. Node B disconnects from its parent and connects with the router becoming the network's new root node. Node C disconnects from the router, enters the idle state, and begins scanning for and selecting a new preferred parent node. **Node C maintains all its downstream connections throughout this process**.

4. Node C selects node B as its preferred parent node, forms an upstream connection, and becomes a second layer node. The network layout is similar after the switch as node C still maintains the same subnetwork. However each node in node C's subnetwork has been placed one layer deeper as a result of the switch. *Parent Node Switching* may adjust the network layout afterwards if any nodes have a new preferred parent node as a result of the root node switch.

Note: Root node switching must require an election hence is only supported when using a self-organized ESP-WIFI-MESH network. In other words, root node switching cannot occur if a designated root node is used.

Parent Node Switching

Parent Node Switching entails a child node switching its upstream connection to another parent node of a shallower layer. **Parent Node Switching occurs autonomously** meaning that a child node will change its upstream connection automatically if a potential parent node of a shallower layer becomes available (i.e. due to a *Asynchronous Power-on Reset*).

All potential parent nodes periodically transmit beacon frames (see *Beacon Frames & RSSI Thresholding*) allowing for a child node to scan for the availability of a shallower parent node. Due to parent node switching, a self-organized ESP-WIFI-MESH network can dynamically adjust its network layout to ensure each connection has a good RSSI and that the number of layers in the network is minimized.

4.10.6 Data Transmission

ESP-WIFI-MESH Packet

ESP-WIFI-MESH network data transmissions use ESP-WIFI-MESH packets. ESP-WIFI-MESH packets are **entirely contained within the frame body of a Wi-Fi data frame**. A multi-hop data transmission in an ESP-WIFI-MESH network will involve a single ESP-WIFI-MESH packet being carried over each wireless hop by a different Wi-Fi data frame.

The following diagram shows the structure of an ESP-WIFI-MESH packet and its relation with a Wi-Fi data frame.

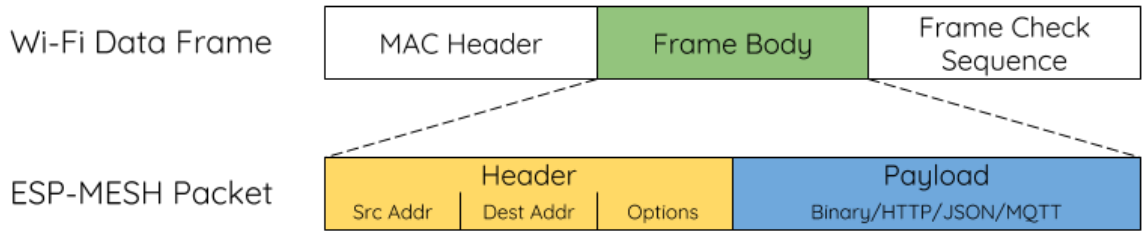


Fig. 33: ESP-WIFI-MESH Packet

The header of an ESP-WIFI-MESH packet contains the MAC addresses of the source and destination nodes. The options field contains information pertaining to the special types of ESP-WIFI-MESH packets such as a group transmission or a packet originating from the external IP network (see `MESH_OPT_SEND_GROUP` and `MESH_OPT_RECV_DS_ADDR`).

The payload of an ESP-WIFI-MESH packet contains the actual application data. This data can be raw binary data, or encoded under an application layer protocol such as HTTP, MQTT, and JSON (see `mesh_proto_t`).

Note: When sending an ESP-WIFI-MESH packet to the external IP network, the destination address field of the header will contain the IP address and port of the target server rather than the MAC address of a node (see `mesh_addr_t`). Furthermore the root node will handle the formation of the outgoing TCP/IP packet.

Group Control & Multicasting

Multicasting is a feature that allows a single ESP-WIFI-MESH packet to be transmitted simultaneously to multiple nodes within the network. Multicasting in ESP-WIFI-MESH can be achieved by either specifying a list

of target nodes, or specifying a preconfigured group of nodes. Both methods of multicasting are called via *esp_mesh_send()*.

To multicast by specifying a list of target nodes, users must first set the ESP-WIFI-MESH packet's destination address to the **Multicast-Group Address** (`01:00:5E:xx:xx:xx`). This signifies that the ESP-WIFI-MESH packet is a multicast packet with a group of addresses, and that the address should be obtained from the header options. Users must then list the MAC addresses of the target nodes as options (see *mesh_opt_t* and *MESH_OPT_SEND_GROUP*). This method of multicasting requires no prior setup but can incur a large amount of overhead data as each target node's MAC address must be listed in the options field of the header.

Multicasting by group allows a ESP-WIFI-MESH packet to be transmitted to a preconfigured group of nodes. Each grouping is identified by a unique ID, and a node can be placed into a group via *esp_mesh_set_group_id()*. Multicasting to a group involves setting the destination address of the ESP-WIFI-MESH packet to the target group ID. Furthermore, the *MESH_DATA_GROUP* flag must set. Using groups to multicast incurs less overhead, but requires nodes to previously added into groups.

Note: During a multicast, all nodes within the network still receive the ESP-WIFI-MESH packet on the MAC layer. However, nodes not included in the MAC address list or the target group will simply filter out the packet.

Broadcasting

Broadcasting is a feature that allows a single ESP-WIFI-MESH packet to be transmitted simultaneously to all nodes within the network. Each node essentially forwards a broadcast packet to all of its upstream and downstream connections such that the packet propagates throughout the network as quickly as possible. However, ESP-WIFI-MESH utilizes the following methods to avoid wasting bandwidth during a broadcast.

1. When an intermediate parent node receives a broadcast packet from its parent, it will forward the packet to each of its child nodes whilst storing a copy of the packet for itself.

2. When an intermediate parent node is the source node of the broadcast, it will transmit the broadcast packet upstream to is parent node and downstream to each of its child nodes.

3. When an intermediate parent node receives a broadcast packet from one of its child nodes, it will forward the packet to its parent node and each of its remaining child nodes whilst storing a copy of the packet for itself.

4. When a leaf node is the source node of a broadcast, it will directly transmit the packet to its parent node.

5. When the root node is the source node of a broadcast, the root node will transmit the packet to all of its child nodes.

6. When the root node receives a broadcast packet from one of its child nodes, it will forward the packet to each of its remaining child nodes whilst storing a copy of the packet for itself.

7. When a node receives a broadcast packet with a source address matching its own MAC address, the node will discard the broadcast packet.

8. When an intermediate parent node receives a broadcast packet from its parent node which was originally transmitted from one of its child nodes, it will discard the broadcast packet

Upstream Flow Control

ESP-WIFI-MESH relies on parent nodes to control the upstream data flow of their immediate child nodes. To prevent a parent node's message buffer from overflowing due to an overload of upstream transmissions, a parent node will allocate a quota for upstream transmissions known as a **receiving window** for each of its child nodes. **Each child node must apply for a receiving window before it is permitted to transmit upstream**. The size of a receiving window can be dynamically adjusted. An upstream transmission from a child node to the parent node consists of the following steps:

1. Before each transmission, the child node sends a window request to its parent node. The window request consists of a sequence number which corresponds to the child node's data packet that is pending transmission.

2. The parent node receives the window request and compares the sequence number with the sequence number of the previous packet sent by the child node. The comparison is used to calculate the size of the receiving window which is transmitted back to the child node.

3. The child node transmits the data packet in accordance with the window size specified by the parent node. If the child node depletes its receiving window, it must obtain another receiving windows by sending a request before it is permitted to continue transmitting.

Note: ESP-WIFI-MESH does not support any downstream flow control.

Warning: Due to *Parent Node Switching*, packet loss may occur during upstream transmissions.

Due to the fact that the root node acts as the sole interface to an external IP network, it is critical that downstream nodes are aware of the root node's connection status with the external IP network. Failing to do so can lead to nodes attempting to pass data upstream to the root node whilst it is disconnected from the IP network. This results in unnecessary transmissions and packet loss. ESP-WIFI-MESH address this issue by providing a mechanism to stabilize the throughput of outgoing data based on the connection status between the root node and the external IP network. The root node can broadcast its external IP network connection status to all other nodes by calling `esp_mesh_post_toDS_state()`.

Bi-Directional Data Stream

The following diagram illustrates the various network layers involved in an ESP-WIFI-MESH Bidirectional Data Stream.

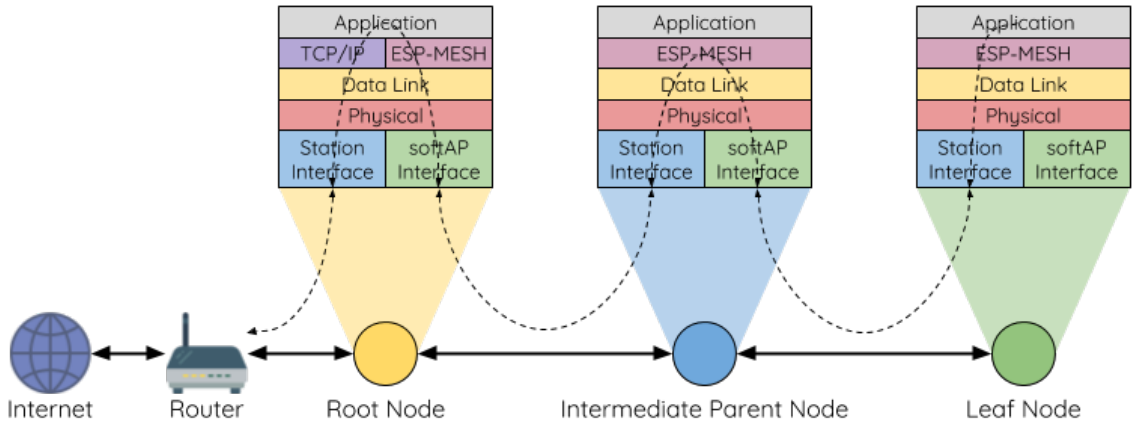


Fig. 34: ESP-WIFI-MESH Bidirectional Data Stream

Due to the use of *Routing Tables*, **ESP-WIFI-MESH is able to handle pack forwarding entirely on the mesh layer**. A TCP/IP layer is only required on the root node when it transmits/receives a packet to/from an external IP network.

4.10.7 Channel Switching

Background

In traditional Wi-Fi networks, **channels** are predetermined frequency ranges. In an infrastructure basic service set (BSS), the serving AP and its connected stations must be on the same operating channels (1 to 14) in which beacons are transmitted. Physically adjacent BSS (Basic Service Sets) operating on the same channel can lead to interference and degraded performance.

In order to allow a BSS adapt to changing physical layer conditions and maintain performance, Wi-Fi contains mechanisms for **network channel switching**. A network channel switch is an attempt to move a BSS to a new operating channel whilst minimizing disruption to the BSS during this process. However it should be recognized that a channel switch may be unsuccessful in moving all stations to the new operating channel.

In an infrastructure Wi-Fi network, network channel switches are triggered by the AP with the aim of having the AP and all connected stations synchronously switch to a new channel. Network channel switching is implemented by embedding a **Channel Switch Announcement (CSA)** element within the AP's periodically transmitted beacon frames. The CSA element is used to advertise to all connected stations regarding an upcoming network channel switch and will be included in multiple beacon frames up until the switch occurs.

A CSA element contains information regarding the **New Channel Number** and a **Channel Switch Count** which indicates the number of beacon frame intervals (TBTTs) remaining until the network channel switch occurs. Therefore, the Channel Switch Count is decremented every beacon frame and allows connected stations to synchronize their channel switch with the AP.

ESP-WIFI-MESH Network Channel Switching

ESP-WIFI-MESH Network Channel Switching also utilize beacon frames that contain a CSA element. However, being a multi-hop network makes the switching process in ESP-WIFI-MESH is more complex due to the fact that a beacon frame might not be able to reach all nodes within the network (i.e. in a single hop). Therefore, an ESP-WIFI-MESH network relies on nodes to forward the CSA element so that it is propagated throughout the network.

When an intermediate parent node with one or more child nodes receives a beacon frame containing a CSA, the node will forward the CSA element by including the element in its next transmitted beacon frame (i.e. with the same **New Channel Number** and **Channel Switch Count**). Given that all nodes within an ESP-WIFI-MESH network receive the same CSA, the nodes can synchronize their channel switches using the Channel Switch Count, albeit with a short delay due to CSA element forwarding.

An ESP-WIFI-MESH network channel switch can be triggered by either the router or the root node.

Root Node Triggered **A root node triggered channel switch can only occur when the ESP-WIFI-MESH network is not connected to a router**. By calling `esp_mesh_switch_channel()`, the root node will set an initial Channel Switch Count value and begin including a CSA element in its beacon frames. Each CSA element is then received by second layer nodes, and forwarded downstream in the their own beacon frames.

Router Triggered When an ESP-WIFI-MESH network is connected to a router, the entire network must use the same channel as the router. Therefore, **the root node will not be permitted to trigger a channel switch when it is connected to a router**.

When the root node receives beacon frame containing a CSA element from the router, **the root node will set Channel Switch Count value in the CSA element to a custom value before forwarding it downstream via beacon frames**. It will also decrement the Channel Switch Count of subsequent CSA elements relative to the custom value. This custom value can be based on factors such as the number of network layers, the current number of nodes etc.

The setting the Channel Switch Count value to a custom value is due to the fact that the ESP-WIFI-MESH network and its router may have a different and varying beacon intervals. Therefore, the Channel Switch Count value provided by the router is irrelevant to an ESP-WIFI-MESH network. By using a custom value, nodes within the ESP-WIFI-MESH network are able to switch channels synchronously relative to the ESP-WIFI-MESH network's beacon interval. However, this will also result in the ESP-WIFI-MESH network's channel switch being unsynchronized with the channel switch of the router and its connected stations.

Impact of Network Channel Switching

- **Due to the ESP-WIFI-MESH network channel switch being unsynchronized with the router's channel switch, there w**
  - The ESP-WIFI-MESH network's channel switch time is dependent on the ESP-WIFI-MESH network's beacon interval and the root node's custom Channel Switch Count value.

  - The channel discrepancy prevents any data exchange between the root node and the router during that ESP-WIFI-MESH network's switch.
  - In the ESP-WIFI-MESH network, the root node and intermediate parent nodes will request their connected child nodes to stop transmissions until the channel switch takes place by setting the **Channel Switch Mode** field in the CSA element to 1.
  - Frequent router triggered network channel switches can degrade the ESP-WIFI-MESH network's performance. Note that this can be caused by the ESP-WIFI-MESH network itself (e.g. due to wireless medium contention with ESP-WIFI-MESH network). If this is the case, users should disable the automatic channel switching on the router and use a specified channel instead.
- **When there is a temporary channel discrepancy, the root node remains technically connected to the router.**
  - Disconnection occurs after the root node fails to receive any beacon frames or probe responses from the router over a fixed number of router beacon intervals.
  - Upon disconnection, the root node will automatically re-scan all channels for the presence of a router.
- **If the root node is unable to receive any of the router's CSA beacon frames (e.g. due to short switch time given by the**
  - After the router switches channels, the root node will no longer be able to receive the router's beacon frames and probe responses and result in a disconnection after a fixed number of beacon intervals.
  - The root node will re-scan all channels for the router after disconnection.
  - The root node will maintain downstream connections throughout this process.

Note: Although ESP-WIFI-MESH network channel switching aims to move all nodes within the network to a new operating channel, it should be recognized that a channel switch might not successfully move all nodes (e.g. due to reasons such as node failures).

Channel and Router Switching Configuration

ESP-WIFI-MESH allows for autonomous channel switching to be enabled/disabled via configuration. Likewise, autonomous router switching (i.e. when a root node autonomously connects to another router) can also be enabled/disabled by configuration. Autonomous channel switching and router switching is dependent on the following configuration parameters and run-time conditions.

Allow Channel Switch: This parameter is set via the `allow_channel_switch` field of the *`mesh_cfg_t`* structure and permits an ESP-WIFI-MESH network to dynamically switch channels when set.

Preset Channel: An ESP-WIFI-MESH network can have a preset channel by setting the `channel` field of the *`mesh_cfg_t`* structure to the desired channel number. If this field is unset, the `allow_channel_switch` parameter is overridden such that channel switches are always permitted.

Allow Router Switch: This parameter is set via the `allow_router_switch` field of the *`mesh_router_t`* and permits an ESP-WIFI-MESH to dynamically switch to a different router when set.

Preset Router BSSID: An ESP-WIFI-MESH network can have a preset router by setting the `bssid` field of the *`mesh_router_t`* structure to the BSSID of the desired router. If this field is unset, the `allow_router_switch` parameter is overridden such that router switches are always permitted.

Root Node Present: The presence of a root node will can also affect whether or a channel or router switch is permitted.

The following table illustrates how the different combinations of parameters/conditions affect whether channel switching and/or router switching is permitted. Note that *X* represents a "don't care" for the parameter.

Preset Channel	Allow Channel Switch	Preset Router BSSID	Allow Router Switch	Root Node Present	Permitted Switches?
N	X	N	X	X	Channel and Router
N	X	Y	N	X	Channel Only
N	X	Y	Y	X	Channel and Router
Y	Y	N	X	X	Channel and Router
Y	N	N	X	N	Router Only
Y	N	N	X	Y	Channel and Router
Y	Y	Y	N	X	Channel Only
Y	N	Y	N	N	N
Y	N	Y	N	Y	Channel Only
Y	Y	Y	Y	X	Channel and Router
Y	N	Y	Y	N	Router Only
Y	N	Y	Y	Y	Channel and Router

4.10.8 Performance

The performance of an ESP-WIFI-MESH network can be evaluated based on multiple metrics such as the following:

Network Building Time: The amount of time taken to build an ESP-WIFI-MESH network from scratch.

Healing Time: The amount of time taken for the network to detect a node break down and carry out appropriate actions to heal the network (such as generating a new root node or forming new connections).

Per-hop latency: The latency of data transmission over one wireless hop. In other words, the time taken to transmit a data packet from a parent node to a child node or vice versa.

Network Node Capacity: The total number of nodes the ESP-WIFI-MESH network can simultaneously support. This number is determined by the maximum number of downstream connections a node can accept and the maximum number of layers permissible in the network.

The following table lists the common performance figures of an ESP-WIFI-MESH network:

- Network Building Time: < 60 seconds
- **Healing time:**
  - Root node break down: < 10 seconds
  - Child node break down: < 5 seconds
- Per-hop latency: 10 to 30 milliseconds

Note: The following test conditions were used to generate the performance figures above.

- Number of test devices: **100**
- Maximum Downstream Connections to Accept: **6**
- Maximum Permissible Layers: **6**

Note: Throughput depends on packet error rate and hop count.

Note: The throughput of root node's access to the external IP network is directly affected by the number of nodes in the ESP-WIFI-MESH network and the bandwidth of the router.

Note: The performance figures can vary greatly between installations based on network configuration and operating environment.

4.10.9 Further Notes

- Data transmission uses Wi-Fi WPA2-PSK encryption
- Mesh networking IE uses AES encryption

Router and internet icon made by Smashicons from www.flaticon.com

4.11 Event Handling

Several ESP-IDF components use *events* to inform application about state changes, such as connection or disconnection. This document gives an overview of these event mechanisms.

4.11.1 Wi-Fi, Ethernet, and IP Events

Before the introduction of *esp_event library*, events from Wi-Fi driver, Ethernet driver, and TCP/IP stack were dispatched using the so-called *legacy event loop*. The following sections explain each of the methods.

esp_event Library Event Loop

esp_event library is designed to supersede the legacy event loop for the purposes of event handling in ESP-IDF. In the legacy event loop, all possible event types and event data structures had to be defined in `system_event_id_t` enumeration and `system_event_info_t` union, which made it impossible to send custom events to the event loop, and use the event loop for other kinds of events (e.g. Mesh). Legacy event loop also supported only one event handler function, therefore application components could not handle some of Wi-Fi or IP events themselves, and required application to forward these events from its event handler function.

See *esp_event library API reference* for general information on using this library. Wi-Fi, Ethernet, and IP events are sent to the *default event loop* provided by this library.

Legacy Event Loop

This event loop implementation is started using `esp_event_loop_init()` function. Application typically supplies an *event handler*, a function with the following signature:

```
esp_err_t event_handler(void *ctx, system_event_t *event)
{
}
```

Both the pointer to event handler function, and an arbitrary context pointer are passed to `esp_event_loop_init()`.

When Wi-Fi, Ethernet, or IP stack generate an event, this event is sent to a high-priority `event` task via a queue. Application-provided event handler function is called in the context of this task. Event task stack size and event queue size can be adjusted using *CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE* and *CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE* options, respectively.

Event handler receives a pointer to the event structure (`system_event_t`) which describes current event. This structure follows a *tagged union* pattern: `event_id` member indicates the type of event, and `event_info` member is a union of description structures. Application event handler will typically use `switch(event->event_id)` to handle different kinds of events.

If application event handler needs to relay the event to some other task, it is important to note that event pointer passed to the event handler is a pointer to temporary structure. To pass the event to another task, application has to make a copy of the entire structure.

Event IDs and Corresponding Data Structures

Event ID (legacy event ID)	Event data structure
Wi-Fi	
WIFI_EVENT_WIFI_READY (SYSTEM_EVENT_WIFI_READY)	n/a
WIFI_EVENT_SCAN_DONE (SYSTEM_EVENT_SCAN_DONE)	*wifi_event_sta_scan_done_t*
WIFI_EVENT_STA_START (SYSTEM_EVENT_STA_START)	n/a
WIFI_EVENT_STA_STOP (SYSTEM_EVENT_STA_STOP)	n/a
WIFI_EVENT_STA_CONNECTED (SYSTEM_EVENT_STA_CONNECTED)	*wifi_event_sta_connected_t*
WIFI_EVENT_STA_DISCONNECTED (SYSTEM_EVENT_STA_DISCONNECTED)	*wifi_event_sta_disconnected_t*
WIFI_EVENT_STA_AUTHMODE_CHANGE (SYSTEM_EVENT_STA_AUTHMODE_CHANGE)	*wifi_event_sta_authmode_change_t*
WIFI_EVENT_STA_WPS_ER_SUCCESS (SYSTEM_EVENT_STA_WPS_ER_SUCCESS)	n/a
WIFI_EVENT_STA_WPS_ER_FAILED (SYSTEM_EVENT_STA_WPS_ER_FAILED)	*wifi_event_sta_wps_fail_reason_t*
WIFI_EVENT_STA_WPS_ER_TIMEOUT (SYSTEM_EVENT_STA_WPS_ER_TIMEOUT)	n/a
WIFI_EVENT_STA_WPS_ER_PIN (SYSTEM_EVENT_STA_WPS_ER_PIN)	*wifi_event_sta_wps_er_pin_t*
WIFI_EVENT_AP_START (SYSTEM_EVENT_AP_START)	n/a
WIFI_EVENT_AP_STOP (SYSTEM_EVENT_AP_STOP)	n/a
WIFI_EVENT_AP_STACONNECTED (SYSTEM_EVENT_AP_STACONNECTED)	*wifi_event_ap_staconnected_t*
WIFI_EVENT_AP_STADISCONNECTED (SYSTEM_EVENT_AP_STADISCONNECTED)	*wifi_event_ap_stadisconnected_t*
WIFI_EVENT_AP_PROBEREQRECVED (SYSTEM_EVENT_AP_PROBEREQRECVED)	*wifi_event_ap_probe_req_rx_t*
Ethernet	
ETHERNET_EVENT_START (SYSTEM_EVENT_ETH_START)	n/a
ETHERNET_EVENT_STOP (SYSTEM_EVENT_ETH_STOP)	n/a
ETHERNET_EVENT_CONNECTED (SYSTEM_EVENT_ETH_CONNECTED)	n/a
ETHERNET_EVENT_DISCONNECTED (SYSTEM_EVENT_ETH_DISCONNECTED)	n/a
IP	
IP_EVENT_STA_GOT_IP (SYSTEM_EVENT_STA_GOT_IP)	*ip_event_got_ip_t*
IP_EVENT_STA_LOST_IP (SYSTEM_EVENT_STA_LOST_IP)	n/a
IP_EVENT_AP_STAIPASSIGNED (SYSTEM_EVENT_AP_STAIPASSIGNED)	n/a
IP_EVENT_GOT_IP6 (SYSTEM_EVENT_GOT_IP6)	*ip_event_got_ip6_t*
IP_EVENT_ETH_GOT_IP (SYSTEM_EVENT_ETH_GOT_IP)	*ip_event_got_ip_t*
IP_EVENT_ETH_LOST_IP (SYSTEM_EVENT_ETH_LOST_IP)	n/a

4.11.2 Mesh Events

ESP-WIFI-MESH uses a system similar to the *Legacy Event Loop* to deliver events to the application. See *System Events* for details.

4.11.3 Bluetooth Events

Various modules of the Bluetooth stack deliver events to applications via dedicated callback functions. Callback functions receive the event type (enumerated value) and event data (union of structures for each event type). The following list gives the registration API name, event enumeration type, and event parameters type.

- BLE GAP: *esp_ble_gap_register_callback()*, *esp_gap_ble_cb_event_t*, *esp_ble_gap_cb_param_t*.
- BT GAP: *esp_bt_gap_register_callback()*, *esp_bt_gap_cb_event_t*, *esp_bt_gap_cb_param_t*.
- GATTC: *esp_ble_gattc_register_callback()*, esp_ble_gattc_cb_event_t, *esp_ble_gattc_cb_param_t*.
- GATTS: *esp_ble_gatts_register_callback()*, esp_ble_gatts_cb_event_t, *esp_ble_gatts_cb_param_t*.
- SPP: *esp_spp_register_callback()*, *esp_spp_cb_event_t*, *esp_spp_cb_param_t*.
- Blufi: *esp_blufi_register_callbacks()*, *esp_blufi_cb_event_t*, *esp_blufi_cb_param_t*.
- A2DP: *esp_a2d_register_callback()*, *esp_a2d_cb_event_t*, *esp_a2d_cb_param_t*.
- AVRC: *esp_avrc_ct_register_callback()*, *esp_avrc_ct_cb_event_t*, *esp_avrc_ct_cb_param_t*.
- HFP Client: *esp_hf_client_register_callback()*, *esp_hf_client_cb_event_t*, *esp_hf_client_cb_param_t*.
- HFP AG: esp_bt_hf_register_callback(), *esp_hf_cb_event_t*, *esp_hf_cb_param_t*.

4.12 Fatal Errors

4.12.1 Overview

In certain situations, execution of the program can not be continued in a well defined way. In ESP-IDF, these situations include:

- CPU Exceptions: Illegal Instruction, Load/Store Alignment Error, Load/Store Prohibited error, Double Exception.
- System level checks and safeguards:
  - *Interrupt watchdog* timeout
  - *Task watchdog* timeout (only fatal if *CONFIG_ESP_TASK_WDT_PANIC* is set)
  - Cache access error
  - Brownout detection event
  - Stack overflow
  - Stack smashing protection check
  - Heap integrity check
  - Undefined behavior sanitizer (UBSAN) checks
- Failed assertions, via `assert`, `configASSERT` and similar macros.

This guide explains the procedure used in ESP-IDF for handling these errors, and provides suggestions on troubleshooting the errors.

4.12.2 Panic Handler

Every error cause listed in the *Overview* will be handled by the *panic handler*.

The panic handler will start by printing the cause of the error to the console. For CPU exceptions, the message will be similar to

```
Guru Meditation Error: Core 0 panic'ed (IllegalInstruction). Exception was␣
↪unhandled.
```

For some of the system level checks (interrupt watchdog, cache access error), the message will be similar to

```
Guru Meditation Error: Core 0 panic'ed (Cache disabled but cached memory␣
↪region accessed). Exception was unhandled.
```

In all cases, the error cause will be printed in parentheses. See *Guru Meditation Errors* for a list of possible error causes.

Subsequent behavior of the panic handler can be set using *CONFIG_ESP_SYSTEM_PANIC* configuration choice. The available options are:

- Print registers and reboot (`CONFIG_ESP_SYSTEM_PANIC_PRINT_REBOOT`) —default option.
  This will print register values at the point of the exception, print the backtrace, and restart the chip.
- Print registers and halt (`CONFIG_ESP_SYSTEM_PANIC_PRINT_HALT`)
  Similar to the above option, but halt instead of rebooting. External reset is required to restart the program.
- Silent reboot (`CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT`)
  Don't print registers or backtrace, restart the chip immediately.
- Invoke GDB Stub (`CONFIG_ESP_SYSTEM_PANIC_GDBSTUB`)
  Start GDB server which can communicate with GDB over console UART port. This option will only provide read-only debugging or post-mortem debugging. See *GDB Stub* for more details.
- Invoke dynamic GDB Stub (`ESP_SYSTEM_GDBSTUB_RUNTIME`)
  Start GDB server which can communicate with GDB over console UART port. This option allows the user to debug a program at run time and set break points, alter the execution, etc. See *GDB Stub* for more details.

The behavior of the panic handler is affected by two other configuration options.

- If *CONFIG_ESP_DEBUG_OCDAWARE* is enabled (which is the default), the panic handler will detect whether a JTAG debugger is connected. If it is, execution will be halted and control will be passed to the debugger. In this case, registers and backtrace are not dumped to the console, and GDBStub / Core Dump functions are not used.
- If the *Core Dump* feature is enabled, then the system state (task stacks and registers) will be dumped to either Flash or UART, for later analysis.
- If *CONFIG_ESP_PANIC_HANDLER_IRAM* is disabled (disabled by default), the panic handler code is placed in flash memory, not IRAM. This means that if ESP-IDF crashes while flash cache is disabled, the panic handler will automatically re-enable flash cache before running GDB Stub or Core Dump. This adds some minor risk, if the flash cache status is also corrupted during the crash.
  If this option is enabled, the panic handler code (including required UART functions) is placed in IRAM, and hence will decrease the usable memory space in SRAM. But this may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash) or when flash cache is corrupted when an exception is triggered.

The following diagram illustrates the panic handler behavior:

4.12.3 Register Dump and Backtrace

Unless the `CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT` option is enabled, the panic handler prints some of the CPU registers, and the backtrace, to the console

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      :␣
↪0x3ffb5030
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      :␣
↪0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      :␣
↪0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     :␣
↪0x06ff1ff8
A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE:␣
↪0x0000001d
```

(continues on next page)

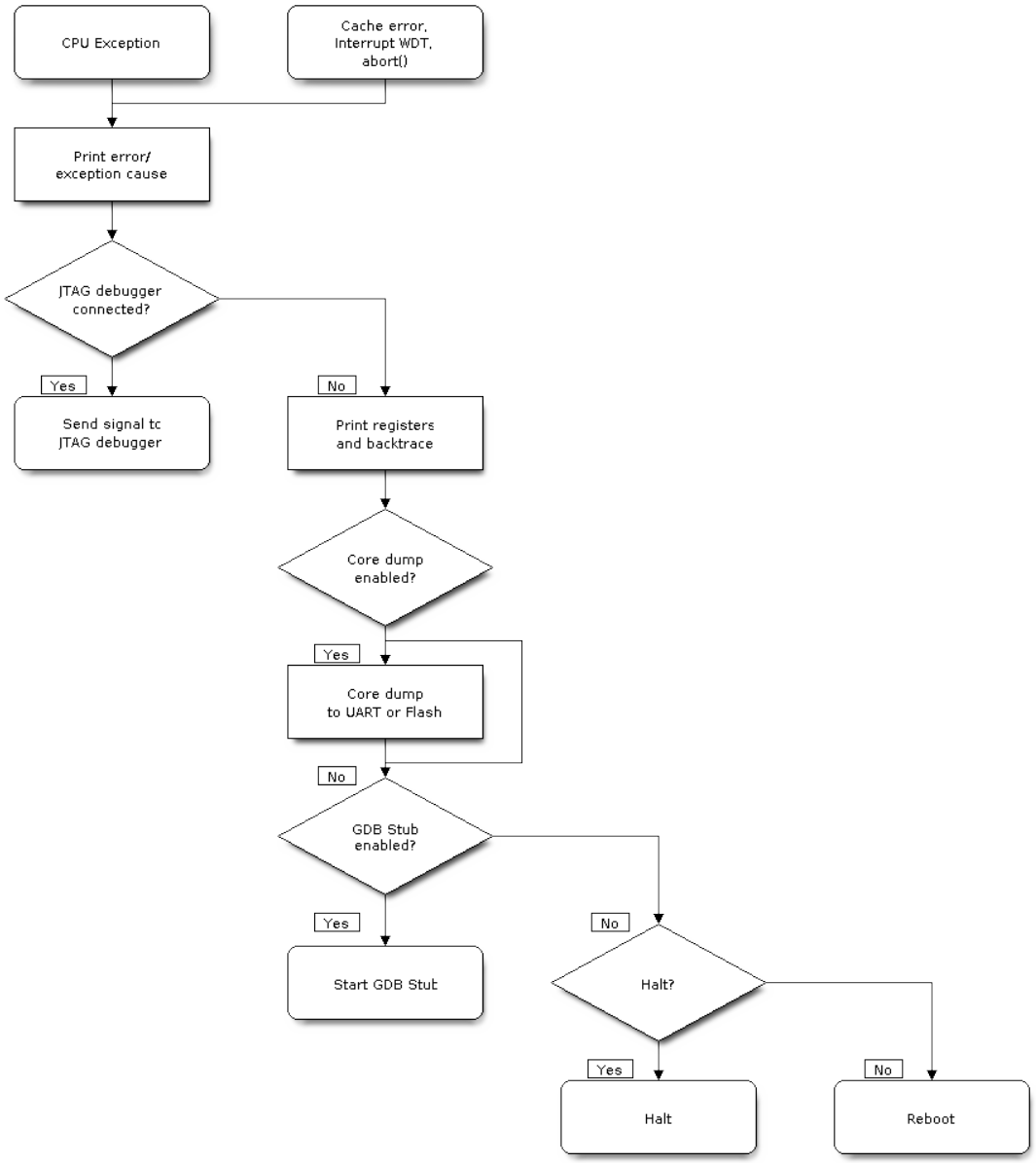


Fig. 35: Panic Handler Flowchart (click to enlarge)

(continued from previous page)

```
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
```

The register values printed are the register values in the exception frame, i.e., values at the moment when the CPU exception or another fatal error has occurred.

A Register dump is not printed if the panic handler has been executed as a result of an `abort()` call.

In some cases, such as interrupt watchdog timeout, the panic handler may print additional CPU registers (EPC1-EPC4) and the registers/backtrace of the code running on the other CPU.

The backtrace line contains PC:SP pairs, where PC is the Program Counter and SP is Stack Pointer, for each stack frame of the current task. If a fatal error happens inside an ISR, the backtrace may include PC:SP pairs both from the task which was interrupted, and from the ISR.

If *IDF Monitor* is used, Program Counter values will be converted to code locations (function name, file name, and line number), and the output will be annotated with additional lines:

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      :␣
↪0x3ffb5030
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      :␣
↪0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      :␣
↪0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     :␣
↪0x06ff1ff8
0x40082d1c: _calloc_r at /Users/user/esp/esp-idf/components/newlib/syscalls.c:51

A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE:␣
↪0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

0x400d0802: main_task at /Users/user/esp/esp-idf/components/esp32/cpu_start.c:470
```

To find the location where a fatal error has happened, look at the lines which follow the "Backtrace" line. Fatal error location is the top line, and subsequent lines show the call stack.

4.12.4 GDB Stub

If the `CONFIG_ESP_SYSTEM_PANIC_GDBSTUB` option is enabled, the panic handler will not reset the chip when a fatal error happens. Instead, it will start a GDB remote protocol server, commonly referred to as GDB Stub. When this happens, a GDB instance running on the host computer can be instructed to connect to the ESP32 UART port.

If *IDF Monitor* is used, GDB is started automatically when a GDB Stub prompt is detected on the UART. The output looks like this:

```
Entering gdb stub now.
$T0b#e6GNU gdb (crosstool-NG crosstool-ng-1.22.0-80-gff1f415) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
```

(continues on next page)

(continued from previous page)

```
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_apple-darwin16.3.0 --target=xtensa-
↪esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /Users/user/esp/example/build/example.elf...done.
Remote debugging using /dev/cu.usbserial-31301
0x400e1b41 in app_main ()
    at /Users/user/esp/example/main/main.cpp:36
36      *((int*) 0) = 0;
(gdb)
```

The GDB prompt can be used to inspect CPU registers, local and static variables, and arbitrary locations in memory. It is not possible to set breakpoints, change the PC, or continue execution. To reset the program, exit GDB and perform an external reset: Ctrl-T Ctrl-R in IDF Monitor, or using the external reset button on the development board.

4.12.5 RTC Watchdog Timeout

The RTC watchdog is used in the startup code to keep track of execution time and it also helps to prevent a lock-up caused by an unstable power source. It is enabled by default (see *CONFIG_BOOTLOADER_WDT_ENABLE*). If the execution time is exceeded, the RTC watchdog will restart the system. In this case, the ROM bootloader will print a message with the `RTC Watchdog Timeout` reason for the reboot.

```
rst:0x10 (RTCWDT_RTC_RESET)
```

The RTC watchdog covers the execution time from the first stage bootloader (ROM bootloader) to application startup. It is initially set in the ROM bootloader, then configured in the bootloader with the *CONFIG_BOOTLOADER_WDT_TIME_MS* option (9000 ms by default). During the application initialization stage, it is reconfigured because the source of the slow clock may have changed, and finally disabled right before the `app_main()` call. There is an option *CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE* which prevents the RTC watchdog from being disabled before `app_main`. Instead, the RTC watchdog remains active and must be fed periodically in your application's code.

4.12.6 Guru Meditation Errors

This section explains the meaning of different error causes, printed in parens after the `Guru Meditation Error: Core panic'ed` message.

Note: See the Guru Meditation Wikipedia article for historical origins of "Guru Meditation".

IllegalInstruction

This CPU exception indicates that the instruction which was executed was not a valid instruction. Most common reasons for this error include:

- FreeRTOS task function has returned. In FreeRTOS, if a task function needs to terminate, it should call `vTaskDelete()` and delete itself, instead of returning.
- Failure to read next instruction from SPI flash. This usually happens if:
  - Application has reconfigured the SPI flash pins as some other function (GPIO, UART, etc.). Consult the Hardware Design Guidelines and the datasheet for the chip or module for details about the SPI flash pins.

  - Some external device has accidentally been connected to the SPI flash pins, and has interfered with communication between ESP32 and SPI flash.
- In C++ code, exiting from a non-void function without returning a value is considered to be an undefined behavior. When optimizations are enabled, the compiler will often omit the epilogue in such functions. This most often results in an IllegalInstruction exception. By default, ESP-IDF build system enables `-Werror=return-type` which means that missing return statements are treated as compile time errors. However if the application project disables compiler warnings, this issue might go undetected and the IllegalInstruction exception will occur at run time.

InstrFetchProhibited

This CPU exception indicates that the CPU could not read an instruction because the address of the instruction does not belong to a valid region in instruction RAM or ROM.

Usually, this means an attempt to call a function pointer, which does not point to valid code. `PC` (Program Counter) register can be used as an indicator: it will be zero or will contain a garbage value (not `0x4xxxxxxx`).

LoadProhibited, StoreProhibited

These CPU exceptions happen when an application attempts to read from or write to an invalid memory location. The address which has been written/read is found in the `EXCVADDR` register in the register dump. If this address is zero, it usually means that the application has attempted to dereference a NULL pointer. If this address is close to zero, it usually means that the application has attempted to access a member of a structure, but the pointer to the structure is NULL. If this address is something else (garbage value, not in `0x3fxxxxxx` - `0x6xxxxxxx` range), it likely means that the pointer used to access the data is either not initialized or has been corrupted.

IntegerDivideByZero

Application has attempted to do an integer division by zero.

LoadStoreAlignment

Application has attempted to read or write a memory location, and the address alignment does not match the load/store size. For example, a 32-bit read can only be done from a 4-byte aligned address, and a 16-bit write can only be done to a 2-byte aligned address.

LoadStoreError

This exception may happen in the following cases:

- If the application has attempted to do an 8- or 16- bit read to, or write from, a memory region which only supports 32-bit reads/writes. For example, dereferencing a `char*` pointer to instruction memory (IRAM, IROM) will result in such an error.
- If the application has attempted to write to a read-only memory region, such as IROM or DROM.

Unhandled debug exception

This will usually be followed by a message like:

```
Debug exception reason: Stack canary watchpoint triggered (task_name)
```

This error indicates that the application has written past the end of the stack of the task with name `task_name`. Note that not every stack overflow is guaranteed to trigger this error. It is possible that the task writes to memory beyond the stack canary location, in which case the watchpoint will not be triggered.

Interrupt wdt timeout on CPU0 / CPU1

Indicates that an interrupt watchdog timeout has occurred. See *Watchdogs* for more information.

Cache disabled but cached memory region accessed

In some situations, ESP-IDF will temporarily disable access to external SPI Flash and SPI RAM via caches. For example, this happens when spi_flash APIs are used to read/write/erase/mmap regions of SPI Flash. In these situations, tasks are suspended, and interrupt handlers not registered with `ESP_INTR_FLAG_IRAM` are disabled. Make sure that any interrupt handlers registered with this flag have all the code and data in IRAM/DRAM. Refer to the *SPI flash API documentation* for more details.

4.12.7 Other Fatal Errors

Brownout

ESP32 has a built-in brownout detector, which is enabled by default. The brownout detector can trigger a system reset if the supply voltage goes below a safe level. The brownout detector can be configured using *CONFIG_ESP_BROWNOUT_DET* and *CONFIG_ESP_BROWNOUT_DET_LVL_SEL* options.

When the brownout detector triggers, the following message is printed:

```
Brownout detector was triggered
```

The chip is reset after the message is printed.

Note that if the supply voltage is dropping at a fast rate, only part of the message may be seen on the console.

Corrupt Heap

ESP-IDF's heap implementation contains a number of run-time checks of the heap structure. Additional checks ("Heap Poisoning") can be enabled in menuconfig. If one of the checks fails, a message similar to the following will be printed:

```
CORRUPT HEAP: Bad tail at 0x3ffe270a. Expected 0xbaad5678 got 0xbaac5678
assertion "head != NULL" failed: file "/Users/user/esp/esp-idf/components/heap/
↪multi_heap_poisoning.c", line 201, function: multi_heap_free
abort() was called at PC 0x400dca43 on core 0
```

Consult *Heap Memory Debugging* documentation for further information.

Stack Smashing

Stack smashing protection (based on GCC `-fstack-protector*` flags) can be enabled in ESP-IDF using *CONFIG_COMPILER_STACK_CHECK_MODE* option. If stack smashing is detected, message similar to the following will be printed:

```
Stack smashing protect failure!

abort() was called at PC 0x400d2138 on core 0

Backtrace: 0x4008e6c0:0x3ffc1780 0x4008e8b7:0x3ffc17a0 0x400d2138:0x3ffc17c0␣
↪0x400e79d5:0x3ffc17e0 0x400e79a7:0x3ffc1840 0x400e79df:0x3ffc18a0␣
↪0x400e2235:0x3ffc18c0 0x400e1916:0x3ffc18f0 0x400e19cd:0x3ffc1910␣
↪0x400e1a11:0x3ffc1930 0x400e1bb2:0x3ffc1950 0x400d2c44:0x3ffc1a80
0
```

The backtrace should point to the function where stack smashing has occurred. Check the function code for unbounded access to local arrays.

Undefined behavior sanitizer (UBSAN) checks

Undefined behavior sanitizer (UBSAN) is a compiler feature which adds run-time checks for potentially incorrect operations, such as:

- overflows (multiplication overflow, signed integer overflow)
- shift base or exponent errors (e.g. shift by more than 32 bits)
- integer conversion errors

See GCC documentation of `-fsanitize=undefined` option for the complete list of supported checks.

Enabling UBSAN UBSAN is disabled by default. It can be enabled at file, component, or project level by adding the `-fsanitize=undefined` compiler option in the build system.

When enabling UBSAN for code which uses the SOC hardware register header files (`soc/xxx_reg.h`), it is recommended to disable shift-base sanitizer using `-fno-sanitize=shift-base` option. This is due to the fact that ESP-IDF register header files currently contain patterns which cause false positives for this specific sanitizer option.

To enable UBSAN at project level, add the following code at the end of the project's `CMakeLists.txt` file:

```
idf_build_set_property(COMPILE_OPTIONS "-fsanitize=undefined" "-fno-sanitize=shift-
↪base" APPEND)
```

Alternatively, pass these options through the `EXTRA_CFLAGS` and `EXTRA_CXXFLAGS` environment variables.

Enabling UBSAN results in significant increase of code and data size. Most applications, except for the trivial ones, will not fit into the available RAM of the microcontroller when UBSAN is enabled for the whole application. Therefore it is recommended that UBSAN is instead enabled for specific components under test.

To enable UBSAN for a specific component (`component_name`) from the project's `CMakeLists.txt` file, add the following code at the end of the file:

```
idf_component_get_property(lib component_name COMPONENT_LIB)
target_compile_options(${lib} PRIVATE "-fsanitize=undefined" "-fno-sanitize=shift-
↪base")
```

Note: See the build system documentation for more information about *build properties* and *component properties*.

To enable UBSAN for a specific component (`component_name`) from `CMakeLists.txt` of the same component, add the following at the end of the file:

```
target_compile_options(${COMPONENT_LIB} PRIVATE "-fsanitize=undefined" "-fno-
↪sanitize=shift-base")
```

UBSAN output When UBSAN detects an error, a message and the backtrace are printed, for example:

```
Undefined behavior of type out_of_bounds

Backtrace:0x4008b383:0x3ffcd8b0 0x4008c791:0x3ffcd8d0 0x4008c587:0x3ffcd8f0␣
↪0x4008c6be:0x3ffcd950 0x400db74f:0x3ffcd970 0x400db99c:0x3ffcd9a0
```

When using *IDF Monitor*, the backtrace will be decoded to function names and source code locations, pointing to the location where the issue has happened (here it is `main.c:128`):

```
0x4008b383: panic_abort at /path/to/esp-idf/components/esp_system/panic.c:367

0x4008c791: esp_system_abort at /path/to/esp-idf/components/esp_system/system_api.
↪c:106

0x4008c587: __ubsan_default_handler at /path/to/esp-idf/components/esp_system/
↪ubsan.c:152

0x4008c6be: __ubsan_handle_out_of_bounds at /path/to/esp-idf/components/esp_system/
↪ubsan.c:223

0x400db74f: test_ub at main.c:128

0x400db99c: app_main at main.c:56 (discriminator 1)
```

The types of errors reported by UBSAN can be as follows:

Name	Meaning
`type_mismatch`, `type_mismatch_v1`	Incorrect pointer value: null, unaligned, not compatible with the given type.
`add_overflow`, `sub_overflow`, `mul_overflow`, `negate_overflow`	Integer overflow during addition, subtraction, multiplication, negation.
`divrem_overflow`	Integer division by 0 or `INT_MIN`.
`shift_out_of_bounds`	Overflow in left or right shift operators.
`out_of_bounds`	Access outside of bounds of an array.
`unreachable`	Unreachable code executed.
`missing_return`	Non-void function has reached its end without returning a value (C++ only).
`vla_bound_not_positive`	Size of variable length array is not positive.
`load_invalid_value`	Value of `bool` or `enum` (C++ only) variable is invalid (out of bounds).
`nonnull_arg`	Null argument passed to a function which is declared with a `nonnull` attribute.
`nonnull_return`	Null value returned from a function which is declared with `returns_nonnull` attribute.
`builtin_unreachable`	`__builtin_unreachable` function called.
`pointer_overflow`	Overflow in pointer arithmetic.

4.13 Flash Encryption

This is a quick start guide to ESP32's flash encryption feature. Using application code as an example, it demonstrates how to test and verify flash encryption operations during development and production.

4.13.1 Introduction

Flash encryption is intended for encrypting the contents of the ESP32's off-chip flash memory. Once this feature is enabled, firmware is flashed as plaintext, and then the data is encrypted in place on the first boot. As a result, physical readout of flash will not be sufficient to recover most flash contents.

With flash encryption enabled, the following types of data are encrypted by default:

- Firmware bootloader
- Partition Table
- All "app" type partitions

Other types of data can be encrypted conditionally:

- Any partition marked with the `encrypted` flag in the partition table. For details, see *Encrypted Partition Flag*.
- Secure Boot bootloader digest if Secure Boot is enabled (see below).

Secure Boot is a separate feature which can be used together with flash encryption to create an even more secure environment.

Important: For production use, flash encryption should be enabled in the "Release" mode only.

Important: Enabling flash encryption limits the options for further updates of ESP32. Before using this feature, read the document and make sure to understand the implications.

4.13.2 Relevant eFuses

The flash encryption operation is controlled by various eFuses available on ESP32. The list of eFuses and their descriptions is given in the table below. The names in eFuse column are also used by espefuse.py tool. For usage in the eFuse API, modify the name by adding `ESP_EFUSE_`, for example: esp_efuse_read_field_bit(ESP_EFUSE_DISABLE_DL_ENCRYPT).

Table 20: eFuses Used in Flash Encryption

eFuse	Description	Bit Depth
`CODING_SCHEME`	Controls actual number of block1 bits used to derive final 256-bit AES key. Possible values: `0` for 256 bits, `1` for 192 bits, `2` for 128 bits. Final AES key is derived based on the `FLASH_CRYPT_CONFIG` value.	2
`flash_encryption` (block1)	AES key storage.	256 bit key block
`FLASH_CRYPT_CONFIG`	Controls the AES encryption process.	4
`DISABLE_DL_ENCRYPT`	If set, disables flash encryption operation while running in Firmware Download mode.	1
`DISABLE_DL_DECRYPT`	If set, disables flash decryption while running in UART Firmware Download mode.	1
`FLASH_CRYPT_CNT`	Enables/disables encryption at boot time. If even number of bits set (0, 2, 4, 6) - encrypt flash at boot time. If odd number of bits set (1, 3, 5, 7) - do not encrypt flash at boot time.	7

Note:

- R/W access control is available for all the eFuse bits listed in the table above.
- The default value of these bits is 0 afer manufacturing.

Read and write access to eFuse bits is controlled by appropriate fields in the registers `WR_DIS` and `RD_DIS`. For more information on ESP32 eFuses, see *eFuse manager*. To change protection bits of eFuse field using espefuse.py, use these two commands: read_protect_efuse and write_protect_efuse. Example `espefuse.py write_protect_efuse DISABLE_DL_ENCRYPT`.

4.13.3 Flash Encryption Process

Assuming that the eFuse values are in their default states and the firmware bootloader is compiled to support flash encryption, the flash encryption process executes as shown below:

1. On the first power-on reset, all data in flash is un-encrypted (plaintext). The ROM bootloader loads the firmware bootloader.
2. Firmware bootloader reads the `FLASH_CRYPT_CNT` eFuse value (`0b0000000`). Since the value is `0` (even number of bits set), it configures and enables the flash encryption block. It also sets the `FLASH_CRYPT_CONFIG` eFuse to 0xF. For more information on the flash encryption block, see *ESP32 Technical Reference Manual > eFuse Controller (eFuse) > Flash Encryption Block* [PDF].
3. Firmware bootloader uses RNG (random) module to generate an AES-256 bit key and then writes it into the `flash_encryption` eFuse. The key cannot be accessed via software as the write and read protection bits for the `flash_encryption` eFuse are set. The flash encryption operations happen entirely by hardware, and the key cannot be accessed via software.
4. Flash encryption block encrypts the flash contents - the firmware bootloader, applications and partitions marked as `encrypted`. Encrypting in-place can take time, up to a minute for large partitions.
5. Firmware bootloader sets the first available bit in `FLASH_CRYPT_CNT` (0b0000001) to mark the flash contents as encrypted. Odd number of bits is set.
6. For *Development Mode*, the firmware bootloader sets only the eFuse bits `DISABLE_DL_DECRYPT` and `DISABLE_DL_CACHE` to allow the UART bootloader to re-flash encrypted binaries. Also, the `FLASH_CRYPT_CNT` eFuse bits are NOT write-protected.
7. For *Release Mode*, the firmware bootloader sets the eFuse bits `DISABLE_DL_ENCRYPT`, `DISABLE_DL_DECRYPT`, and `DISABLE_DL_CACHE` to 1 to prevent the UART bootloader from decrypting the flash contents. It also write-protects the `FLASH_CRYPT_CNT` eFuse bits. To modify this behavior, see *Enabling UART Bootloader Encryption/Decryption*.
8. The device is then rebooted to start executing the encrypted image. The firmware bootloader calls the flash decryption block to decrypt the flash contents and then loads the decrypted contents into IRAM.

During the development stage, there is a frequent need to program different plaintext flash images and test the flash encryption process. This requires that Firmware Download mode is able to load new plaintext images as many times as it might be needed. However, during manufacturing or production stages, Firmware Download mode should not be allowed to access flash contents for security reasons.

Hence, two different flash encryption configurations were created: for development and for production. For details on these configurations, see Section *Flash Encryption Configuration*.

4.13.4 Flash Encryption Configuration

The following flash encryption modes are available:

- *Development Mode* - recommended for use only during development. In this mode, it is still possible to flash new plaintext firmware to the device, and the bootloader will transparently encrypt this firmware using the key stored in hardware. This allows, indirectly, to read out the plaintext of the firmware in flash.
- *Release Mode* - recommended for manufacturing and production. In this mode, flashing plaintext firmware to the device without knowing the encryption key is no longer possible.

This section provides information on the mentioned flash encryption modes and step by step instructions on how to use them.

Development Mode

During development, you can encrypt flash using either an ESP32 generated key or external host-generated key.

Using ESP32 Generated Key Development mode allows you to download multiple plaintext images using Firmware Download mode.

To test flash encryption process, take the following steps:

1. Ensure that you have an ESP32 device with default flash encryption eFuse settings as shown in *Relevant eFuses*.

   See how to check *ESP32 Flash Encryption Status*.

2. In *Project Configuration Menu*, do the following:

- *Enable flash encryption on boot*.
- *Select encryption mode* (**Development mode** by default).
- *Select UART ROM download mode* (**enabled** by default). Note that for the ESP32 target, the choice is only available when *CONFIG_ESP32_REV_MIN* level is set to 3 (ESP32 V3).
- *Select the appropriate bootloader log verbosity*.
- Save the configuration and exit.

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

3. Run the command given below to build and flash the complete images.

```
idf.py flash monitor
```

Note: This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.

This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as `encrypted` then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

A sample output of the first ESP32 boot after enabling flash encryption is given below:

```
--- idf_monitor on /dev/cu.SLAB_USBtoUART 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13608
load:0x40080400,len:6664
entry 0x40080764
I (28) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (29) boot: compile time 15:37:14
I (30) boot: Enabling RNG early entropy source...
I (35) boot: SPI Speed      : 40MHz
I (39) boot: SPI Mode       : DIO
I (43) boot: SPI Flash Size : 4MB
I (47) boot: Partition Table:
I (51) boot: ## Label            Usage          Type ST Offset   Length
I (58) boot:  0 nvs              WiFi data        01 02 0000a000 00006000
I (66) boot:  1 phy_init         RF data          01 01 00010000 00001000
I (73) boot:  2 factory          factory app      00 00 00020000 00100000
I (81) boot: End of partition table
I (85) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x0808c (␣
↪32908) map
I (105) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000 size=0x01ea4 ( ␣
↪7844) load
I (109) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000 size=0x00400 ( ␣
↪1024) load
0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/xtensa_vectors.
↪S:1778
```

(continues on next page)

(continued from previous page)

```
I (114) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400 size=0x05ca8 (␣
↪23720) load
I (132) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018 size=0x126a8 (␣
↪75432) map
0x400d0018: _flash_cache_start at ??:?

I (159) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8 size=0x01f4c ( ␣
↪8012) load
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/freertos/tasks.
↪c:4561

I (168) boot: Loaded app from partition at offset 0x20000
I (168) boot: Checking flash encryption...
I (168) flash_encrypt: Generating new flash encryption key...
I (187) flash_encrypt: Read & write protecting new key...
I (187) flash_encrypt: Setting CRYPT_CONFIG efuse to 0xF
W (188) flash_encrypt: Not disabling UART bootloader encryption
I (195) flash_encrypt: Disable UART bootloader decryption...
I (201) flash_encrypt: Disable UART bootloader MMU cache...
I (208) flash_encrypt: Disable JTAG...
I (212) flash_encrypt: Disable ROM BASIC interpreter fallback...
I (219) esp_image: segment 0: paddr=0x00001020 vaddr=0x3fff0018 size=0x00004 (    ␣
↪4)
I (227) esp_image: segment 1: paddr=0x0000102c vaddr=0x3fff001c size=0x02104 ( ␣
↪8452)
I (239) esp_image: segment 2: paddr=0x00003138 vaddr=0x40078000 size=0x03528 (␣
↪13608)
I (249) esp_image: segment 3: paddr=0x00006668 vaddr=0x40080400 size=0x01a08 ( ␣
↪6664)
I (657) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x0808c (␣
↪32908) map
I (669) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000 size=0x01ea4 ( ␣
↪7844)
I (672) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000 size=0x00400 ( ␣
↪1024)
0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/xtensa_vectors.
↪S:1778

I (676) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400 size=0x05ca8 (␣
↪23720)
I (692) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018 size=0x126a8 (␣
↪75432) map
0x400d0018: _flash_cache_start at ??:?

I (719) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8 size=0x01f4c ( ␣
↪8012)
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/freertos/tasks.
↪c:4561

I (722) flash_encrypt: Encrypting partition 2 at offset 0x20000...
I (13229) flash_encrypt: Flash encryption completed
I (13229) boot: Resetting with flash encryption enabled...
```

A sample output of subsequent ESP32 boots just mentions that flash encryption is already enabled:

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
```

(continues on next page)

(continued from previous page)

```
load:0x3fff001c,len:8452
load:0x40078000,len:13652
ho 0 tail 12 room 4
load:0x40080400,len:6664
entry 0x40080764
I (30) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (30) boot: compile time 16:32:53
I (31) boot: Enabling RNG early entropy source...
I (37) boot: SPI Speed      : 40MHz
I (41) boot: SPI Mode       : DIO
I (45) boot: SPI Flash Size : 4MB
I (49) boot: Partition Table:
I (52) boot: ## Label            Usage          Type ST Offset   Length
I (60) boot:  0 nvs              WiFi data        01 02 0000a000 00006000
I (67) boot:  1 phy_init         RF data          01 01 00010000 00001000
I (75) boot:  2 factory          factory app      00 00 00020000 00100000
I (82) boot: End of partition table
I (86) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x0808c (␣
↪32908) map
I (107) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000 size=0x01ea4 ( ␣
↪7844) load
I (111) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000 size=0x00400 ( ␣
↪1024) load
0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/xtensa_vectors.
↪S:1778

I (116) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400 size=0x05ca8 (␣
↪23720) load
I (134) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018 size=0x126a8 (␣
↪75432) map
0x400d0018: _flash_cache_start at ??:?

I (162) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8 size=0x01f4c ( ␣
↪8012) load
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/freertos/tasks.
↪c:4561

I (171) boot: Loaded app from partition at offset 0x20000
I (171) boot: Checking flash encryption...
I (171) flash_encrypt: flash encryption is enabled (3 plaintext flashes left)
I (178) boot: Disabling RNG early entropy source...
I (184) cpu_start: Pro cpu up.
I (188) cpu_start: Application information:
I (193) cpu_start: Project name:     flash-encryption
I (198) cpu_start: App version:      v4.0-dev-850-gc4447462d-dirty
I (205) cpu_start: Compile time:     Jun 17 2019 16:32:52
I (211) cpu_start: ELF file SHA256:  8770c886bdf561a7...
I (217) cpu_start: ESP-IDF:          v4.0-dev-850-gc4447462d-dirty
I (224) cpu_start: Starting app cpu, entry point is 0x40080e4c
0x40080e4c: call_start_cpu1 at esp-idf/esp-idf/components/esp32/cpu_start.c:265

I (0) cpu_start: App cpu up.
I (235) heap_init: Initializing. RAM available for dynamic allocation:
I (241) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (247) heap_init: At 3FFB2EC8 len 0002D138 (180 KiB): DRAM
I (254) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (260) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (266) heap_init: At 40087FF4 len 0001800C (96 KiB): IRAM
I (273) cpu_start: Pro cpu start user code
I (291) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
```

(continues on next page)

(continued from previous page)

```
Sample program to check Flash Encryption
This is ESP32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 4MB external␣
↪flash
Flash encryption feature is enabled
Flash encryption mode is DEVELOPMENT
Flash in encrypted mode with flash_crypt_cnt = 1
Halting...
```

At this stage, if you need to update and re-flash binaries, see *Re-flashing Updated Partitions*.

Using Host Generated Key It is possible to pre-generate a flash encryption key on the host computer and burn it into the eFuse. This allows you to pre-encrypt data on the host and flash already encrypted data without needing a plaintext flash update. This feature can be used in both *Development Mode* and *Release Mode*. Without a pre-generated key, data is flashed in plaintext and then ESP32 encrypts the data in-place.

Note: This option is not recommended for production, unless a separate key is generated for each individual device.

To use a host generated key, take the following steps:

1. Ensure that you have an ESP32 device with default flash encryption eFuse settings as shown in *Relevant eFuses*.

   See how to check *ESP32 Flash Encryption Status*.

2. Generate a random key by running:

   ```
   espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
   ```

3. **Before the first encrypted boot**, burn the key into your device's eFuse using the command below. This action can be done **only once**.

   ```
   espefuse.py --port PORT burn_key flash_encryption my_flash_encryption_key.
   ↪bin
   ```

   If the key is not burned and the device is started after enabling flash encryption, the ESP32 will generate a random key that software cannot access or modify.

4. In *Project Configuration Menu*, do the following:
   - *Enable flash encryption on boot*
   - *Select encryption mode* (**Development mode** by default)
   - *Select the appropriate bootloader log verbosity*
   - Save the configuration and exit.

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

5. Run the command given below to build and flash the complete images.

   ```
   idf.py flash monitor
   ```

   Note: This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.

   This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as

`encrypted` then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

If using Development Mode, then the easiest way to update and re-flash binaries is *Re-flashing Updated Partitions*.

If using Release Mode, then it is possible to pre-encrypt the binaries on the host and then flash them as ciphertext. See *Manually Encrypting Files*.

Re-flashing Updated Partitions If you update your application code (done in plaintext) and want to re-flash it, you will need to encrypt it before flashing. To encrypt the application and flash it in one step, run:

```
idf.py encrypted-app-flash monitor
```

If all partitions needs to be updated in encrypted format, run:

```
idf.py encrypted-flash monitor
```

Release Mode

In Release mode, UART bootloader cannot perform flash encryption operations. New plaintext images can ONLY be downloaded using the over-the-air (OTA) scheme which will encrypt the plaintext image before writing to flash.

To use this mode, take the following steps:

1. Ensure that you have an ESP32 device with default flash encryption eFuse settings as shown in *Relevant eFuses*.

   See how to check *ESP32 Flash Encryption Status*.

2. In *Project Configuration Menu*, do the following:

   - *Enable flash encryption on boot*
   - *Select Release mode* (Note that once Release mode is selected, the `DISABLE_DL_ENCRYPT` and `DISABLE_DL_DECRYPT` eFuse bits will be burned to disable flash encryption hardware in ROM Download Mode.)
   - *Select UART ROM download mode (Permanently disabled (recommended))* (Note that this option is only available when *CONFIG_ESP32_REV_MIN* is set to 3 (ESP32 V3).) The default choice is to keep UART ROM download mode enabled, however it' s recommended to permanently disable this mode to reduce the options available to an attacker.
   - *Select the appropriate bootloader log verbosity*
   - Save the configuration and exit.

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

3. Run the command given below to build and flash the complete images.

   ```
   idf.py flash monitor
   ```

   Note: This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.

   This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as `encrypted` then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

Once the flash encryption is enabled in Release mode, the bootloader will write-protect the FLASH_CRYPT_CNT eFuse.

For subsequent plaintext field updates, use *OTA scheme*.

Note: If you have pre-generated the flash encryption key and stored a copy, and the UART download mode is not permanently disabled via *CONFIG_SECURE_UART_ROM_DL_MODE* (ESP32 V3 only), then it is possible to update the flash locally by pre-encrypting the files and then flashing the ciphertext. See *Manually Encrypting Files*.

Best Practices

When using Flash Encryption in production:

- Do not reuse the same flash encryption key between multiple devices. This means that an attacker who copies encrypted data from one device cannot transfer it to a second device.
- When using ESP32 V3, if the UART ROM Download Mode is not needed for a production device then it should be disabled to provide an extra level of protection. Do this by calling `esp_efuse_disable_rom_download_mode()` during application startup. Alternatively, configure the project *CONFIG_ESP32_REV_MIN* level to 3 (targeting ESP32 V3 only) and select the *CONFIG_SECURE_UART_ROM_DL_MODE* to "Permanently disable ROM Download Mode (recommended)". The ability to disable ROM Download Mode is not available on earlier ESP32 versions.
- Enable *Secure Boot* as an extra layer of protection, and to prevent an attacker from selectively corrupting any part of the flash before boot.

4.13.5 Possible Failures

Once flash encryption is enabled, the FLASH_CRYPT_CNT eFuse value will have an odd number of bits set. It means that all the partitions marked with the encryption flag are expected to contain encrypted ciphertext. Below are the three typical failure cases if the ESP32 is erroneously loaded with plaintext data:

1. If the bootloader partition is re-flashed with a **plaintext firmware bootloader image**, the ROM bootloader will fail to load the firmware bootloader resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57
```

Note: This error also appears if the flash contents are erased or corrupted.

2. If the firmware bootloader is encrypted, but the partition table is re-flashed with a **plaintext partition table image**, the bootloader will fail to read the partition table resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:10464
ho 0 tail 12 room 4
load:0x40078000,len:19168
load:0x40080400,len:6664
entry 0x40080764
I (60) boot: ESP-IDF v4.0-dev-763-g2c55fae6c-dirty 2nd stage bootloader
I (60) boot: compile time 19:15:54
I (62) boot: Enabling RNG early entropy source...
I (67) boot: SPI Speed      : 40MHz
I (72) boot: SPI Mode       : DIO
I (76) boot: SPI Flash Size : 4MB
E (80) flash_parts: partition 0 invalid magic number 0x94f6
E (86) boot: Failed to verify partition table
E (91) boot: load partition table error!
```

3. If the bootloader and partition table are encrypted, but the application is re-flashed with a **plaintext application image**, the bootloader will fail to load the application resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13616
load:0x40080400,len:6664
entry 0x40080764
I (56) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (56) boot: compile time 15:37:14
I (58) boot: Enabling RNG early entropy source...
I (64) boot: SPI Speed      : 40MHz
I (68) boot: SPI Mode       : DIO
I (72) boot: SPI Flash Size : 4MB
I (76) boot: Partition Table:
I (79) boot: ## Label            Usage          Type ST Offset   Length
I (87) boot:  0 nvs              WiFi data        01 02 0000a000 00006000
I (94) boot:  1 phy_init         RF data          01 01 00010000 00001000
I (102) boot:  2 factory          factory app      00 00 00020000 00100000
I (109) boot: End of partition table
E (113) esp_image: image at 0x20000 has invalid magic byte
W (120) esp_image: image at 0x20000 has invalid SPI mode 108
W (126) esp_image: image at 0x20000 has invalid SPI size 11
E (132) boot: Factory app partition is not bootable
E (138) boot: No bootable app partitions in the partition table
```

4.13.6 ESP32 Flash Encryption Status

1. Ensure that you have an ESP32 device with default flash encryption eFuse settings as shown in *Relevant eFuses*.

To check if flash encryption on your ESP32 device is enabled, do one of the following:

- flash the application example security/flash_encryption onto your device. This application prints the FLASH_CRYPT_CNT eFuse value and if flash encryption is enabled or disabled.
- *Find the serial port name* under which your ESP32 device is connected, replace PORT with your port name in the following command, and run it:

```
espefuse.py -p PORT summary
```

4.13.7 Reading and Writing Data in Encrypted Flash

ESP32 application code can check if flash encryption is currently enabled by calling *esp_flash_encryption_enabled()*. Also, a device can identify the flash encryption mode by calling *esp_get_flash_encryption_mode()*.

Once flash encryption is enabled, be more careful with accessing flash contents from code.

Scope of Flash Encryption

Whenever the FLASH_CRYPT_CNT eFuse is set to a value with an odd number of bits, all flash content accessed via the MMU's flash cache is transparently decrypted. It includes:

- Executable application code in flash (IROM).
- All read-only data stored in flash (DROM).
- Any data accessed via *spi_flash_mmap()*.
- The firmware bootloader image when it is read by the ROM bootloader.

Important: The MMU flash cache unconditionally decrypts all existing data. Data which is stored unencrypted in flash memory will also be "transparently decrypted" via the flash cache and will appear to software as random garbage.

Reading from Encrypted Flash

To read data without using a flash cache MMU mapping, you can use the partition read function *esp_partition_read()*. This function will only decrypt data when it is read from an encrypted partition. Data read from unencrypted partitions will not be decrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

You can also use the following SPI flash API functions:

- *esp_flash_read()* to read raw (encrypted) data which will not be decrypted
- *esp_flash_read_encrypted()* to read and decrypt data

Data stored using the Non-Volatile Storage (NVS) API is always stored and read decrypted from the perspective of flash encryption. It is up to the library to provide encryption feature if required. Refer to *NVS Encryption* for more details.

Writing to Encrypted Flash

It is recommended to use the partition write function *esp_partition_write()*. This function will only encrypt data when it is written to an encrypted partition. Data written to unencrypted partitions will not be encrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

You can also pre-encrypt and write data using the function *esp_flash_write_encrypted()*

Also, the following ROM function exist but not supported in esp-idf applications:

- esp_rom_spiflash_write_encrypted pre-encrypts and writes data to flash
- SPIWrite writes unencrypted data to flash

Since data is encrypted in blocks, the minimum write size for encrypted data is 16 bytes and the alignment is also 16 bytes.

4.13.8 Updating Encrypted Flash

OTA Updates

OTA updates to encrypted partitions will automatically write encrypted data if the function `esp_partition_write()` is used.

Before building the application image for OTA updating of an already encrypted device, enable the option *Enable flash encryption on boot* in project configuration menu.

For general information about ESP-IDF OTA updates, please refer to *OTA*

Updating Encrypted Flash via Serial

Flashing an encrypted device via serial bootloader requires that the serial bootloader download interface has not been permanently disabled via eFuse.

In Development Mode, the recommended method is *Re-flashing Updated Partitions*.

In Release Mode, if a copy of the same key stored in eFuse is available on the host then it's possible to pre-encrypt files on the host and then flash them. See *Manually Encrypting Files*.

4.13.9 Disabling Flash Encryption

If flash encryption was enabled accidentally, flashing of plaintext data will soft-brick the ESP32. The device will reboot continuously, printing the error `flash read err, 1000` or `invalid header: 0xXXXXXX`.

For flash encryption in Development mode, encryption can be disabled by burning the `FLASH_CRYPT_CNT` eFuse. It can only be done three times per chip by taking the following steps:

1. In *Project Configuration Menu*, disable *Enable flash encryption on boot*, then save and exit.
2. Open project configuration menu again and **double-check** that you have disabled this option! If this option is left enabled, the bootloader will immediately re-enable encryption when it boots.
3. With flash encryption disabled, build and flash the new bootloader and application by running `idf.py flash`.
4. Use `espefuse.py` (in `components/esptool_py/esptool`) to disable the `FLASH_CRYPT_CNT` by running:

```
espefuse.py burn_efuse FLASH_CRYPT_CNT
```

Reset the ESP32. Flash encryption will be disabled, and the bootloader will boot as usual.

4.13.10 Key Points About Flash Encryption

- Flash memory contents is encrypted using AES-256. The flash encryption key is stored in the `flash_encryption` eFuse internal to the chip and, by default, is protected from software access.
- The flash encryption algorithm is AES-256, where the key is "tweaked" with the offset address of each 32 byte block of flash. This means that every 32-byte block (two consecutive 16 byte AES blocks) is encrypted with a unique key derived from the flash encryption key.
- Flash access is transparent via the flash cache mapping feature of ESP32 - any flash regions which are mapped to the address space will be transparently decrypted when read.
  Some data partitions might need to remain unencrypted for ease of access or might require the use of flash-friendly update algorithms which are ineffective if the data is encrypted. NVS partitions for non-volatile storage

cannot be encrypted since the NVS library is not directly compatible with flash encryption. For details, refer to *NVS Encryption*.
- If flash encryption might be used in future, the programmer must keep it in mind and take certain precautions when writing code that *uses encrypted flash*.
- If secure boot is enabled, re-flashing the bootloader of an encrypted device requires a "Re-flashable" secure boot digest (see *Flash Encryption and Secure Boot*).

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

Important: Do not interrupt power to the ESP32 while the first boot encryption pass is running. If power is interrupted, the flash contents will be corrupted and will require flashing with unencrypted data again. In this case, re-flashing will not count towards the flashing limit.

4.13.11 Limitations of Flash Encryption

Flash encryption protects firmware against unauthorised readout and modification. It is important to understand the limitations of the flash encryption feature:

- Flash encryption is only as strong as the key. For this reason, we recommend keys are generated on the device during first boot (default behaviour). If generating keys off-device, ensure proper procedure is followed and don't share the same key between all production devices.
- Not all data is stored encrypted. If storing data on flash, check if the method you are using (library, API, etc.) supports flash encryption.
- Flash encryption does not prevent an attacker from understanding the high-level layout of the flash. This is because the same AES key is used for every pair of adjacent 16 byte AES blocks. When these adjacent 16 byte blocks contain identical content (such as empty or padding areas), these blocks will encrypt to produce matching pairs of encrypted blocks. This may allow an attacker to make high-level comparisons between encrypted devices (i.e. to tell if two devices are probably running the same firmware version).
- For the same reason, an attacker can always tell when a pair of adjacent 16 byte blocks (32 byte aligned) contain two identical 16 byte sequences. Keep this in mind if storing sensitive data on the flash, design your flash storage so this doesn't happen (using a counter byte or some other non-identical value every 16 bytes is sufficient). *NVS Encryption* deals with this and is suitable for many uses.
- Flash encryption alone may not prevent an attacker from modifying the firmware of the device. To prevent unauthorised firmware from running on the device, use flash encryption in combination with *Secure Boot*.

4.13.12 Flash Encryption and Secure Boot

It is recommended to use flash encryption in combination with Secure Boot. However, if Secure Boot is enabled, additional restrictions apply to device re-flashing:

- *OTA Updates* are not restricted, provided that the new app is signed correctly with the Secure Boot signing key.
- *Plaintext serial flash updates* are only possible if the *Re-flashable* Secure Boot mode is selected and a Secure Boot key was pre-generated and burned to the ESP32 (refer to *Secure Boot*). In such configuration, `idf.py bootloader` will produce a pre-digested bootloader and secure boot digest file for flashing at offset 0x0. When following the plaintext serial re-flashing steps it is necessary to re-flash this file before flashing other plaintext data.
- *Re-flashing via Pregenerated Flash Encryption Key* is still possible, provided the bootloader is not re-flashed. Re-flashing the bootloader requires the same *Re-flashable* option to be enabled in the Secure Boot config.

4.13.13 Advanced Features

The following section covers advanced features of flash encryption.

Encrypted Partition Flag

Some partitions are encrypted by default. Other partitions can be marked in the partition table description as requiring encryption by adding the flag `encrypted` to the partitions' flag field. As a result, data in these marked partitions will be treated as encrypted in the same manner as an app partition.

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000, 1M
secret_data, 0x40, 0x01, 0x20000, 256K, encrypted
```

For details on partition table description, see *partition table*.

Further information about encryption of partitions:

- Default partition tables do not include any encrypted data partitions.
- With flash encryption enabled, the `app` partition is always treated as encrypted and does not require marking.
- If flash encryption is not enabled, the flag "encrypted" has no effect.
- You can also consider protecting `phy_init` data from physical access, readout, or modification, by marking the optional `phy` partition with the flag `encrypted`.
- The `nvs` partition cannot be encrypted, because the NVS library is not directly compatible with flash encryption.

Enabling UART Bootloader Encryption/Decryption

On the first boot, the flash encryption process burns by default the following eFuses:

- `DISABLE_DL_ENCRYPT` which disables flash encryption operation when running in UART bootloader boot mode.
- `DISABLE_DL_DECRYPT` which disables transparent flash decryption when running in UART bootloader mode, even if the eFuse `FLASH_CRYPT_CNT` is set to enable it in normal operation.
- `DISABLE_DL_CACHE` which disables the entire MMU flash cache when running in UART bootloader mode.

However, before the first boot you can choose to keep any of these features enabled by burning only selected eFuses and write-protect the rest of eFuses with unset value 0. For example:

```
espefuse.py --port PORT burn_efuse DISABLE_DL_DECRYPT
espefuse.py --port PORT write_protect_efuse DISABLE_DL_ENCRYPT
```

Important: Leaving `DISABLE_DL_DECRYPT` unset (0) makes flash encryption useless.

An attacker with physical access to the chip can use UART bootloader mode with custom stub code to read out the flash contents.

Setting FLASH_CRYPT_CONFIG

The eFuse `FLASH_CRYPT_CONFIG` determines the number of bits in the flash encryption key which are "tweaked" with the block offset. For details, see *Flash Encryption Algorithm*.

On the first boot of the firmware bootloader, this value is set to the maximum `0xF`.

It is possible to burn this eFuse manually and write protect it before the first boot in order to select different tweak values. However, this is not recommended.

It is strongly recommended to never write-protect `FLASH_CRYPT_CONFIG` when it is unset. Otherwise, its value will remain zero permanently, and no bits in the flash encryption key will be tweaked. As a result, the flash encryption algorithm will be equivalent to AES ECB mode.

JTAG Debugging

By default, when Flash Encryption is enabled (in either Development or Release mode) then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables flash encryption.

See *JTAG with Flash Encryption or Secure Boot* for more information about using JTAG Debugging with Flash Encryption.

Manually Encrypting Files

Manually encrypting or decrypting files requires the flash encryption key to be pre-burned in eFuse (see *Using Host Generated Key*) and a copy to be kept on the host. If the flash encryption is configured in Development Mode then it's not necessary to keep a copy of the key or follow these steps, the simpler *Re-flashing Updated Partitions* steps can be used.

The key file should be a single raw binary file (example: `key.bin`).

For example, these are the steps to encrypt the file `build/my-app.bin` to flash at offset 0x10000. Run espsecure.py as follows:

```
espsecure.py encrypt_flash_data --keyfile /path/to/key.bin --address 0x10000 --
↪output my-app-ciphertext.bin build/my-app.bin
```

The file `my-app-ciphertext.bin` can then be flashed to offset 0x10000 using `esptool.py`. To see all of the command line options recommended for `esptool.py`, see the output printed when `idf.py build` succeeds.

Note: If the flashed ciphertext file is not recognized by the ESP32 when it boots, check that the keys match and that the command line arguments match exactly, including the correct offset.

If your ESP32 uses non-default *FLASH_CRYPT_CONFIG value in eFuse* then you will need to pass the `--flash_crypt_conf` argument to `espsecure.py` to set the matching value. This will not happen if the device configured flash encryption by itself, but may happen if burning eFuses manually to enable flash encryption.

The command `espsecure.py decrypt_flash_data` can be used with the same options (and different input/output files), to decrypt ciphertext flash contents or a previously encrypted file.

4.13.14 Technical Details

The following sections provide some reference information about the operation of flash encryption.

Flash Encryption Algorithm

- AES-256 operates on 16-byte blocks of data. The flash encryption engine encrypts and decrypts data in 32-byte blocks - two AES blocks in series.
- The main flash encryption key is stored in the `flash_encryption` eFuse and, by default, is protected from further writes or software readout.
- AES-256 key size is 256 bits (32 bytes) read from the `flash_encryption` eFuse. The hardware AES engine uses the key in reversed byte order as compared to the storage order in `flash_encryption`.
  - If the `CODING_SCHEME` eFuse is set to `0` (default, "None" Coding Scheme) then the eFuse key block is 256 bits and the key is stored as-is (in reversed byte order).
  - If the `CODING_SCHEME` eFuse is set to `1` (3/4 Encoding) then the eFuse key block is 192 bits (in reversed byte order), so overall entropy is reduced. The hardware flash encryption still operates on a 256-bit key, after being read (and un-reversed), the key is extended as `key = key[0:255] + key[64:127]`.
- AES algorithm is used inverted in flash encryption, so the flash encryption "encrypt" operation is AES decrypt and the "decrypt" operation is AES encrypt. This is for performance reasons and does not alter the effeciency of the algorithm.

- Each 32-byte block (two adjacent 16-byte AES blocks) is encrypted with a unique key. The key is derived from the main flash encryption key in `flash_encryption`, XORed with the offset of this block in the flash (a "key tweak").
- The specific tweak depends on the `FLASH_CRYPT_CONFIG` eFuse setting. This is a 4-bit eFuse where each bit enables XORing of a particular range of the key bits:
  - Bit 1, bits 0-66 of the key are XORed.
  - Bit 2, bits 67-131 of the key are XORed.
  - Bit 3, bits 132-194 of the key are XORed.
  - Bit 4, bits 195-256 of the key are XORed.

  It is recommended that `FLASH_CRYPT_CONFIG` is always left at the default value `0xF`, so that all key bits are XORed with the block offset. For details, see *Setting FLASH_CRYPT_CONFIG*.
- The high 19 bits of the block offset (bit 5 to bit 23) are XORed with the main flash encryption key. This range is chosen for two reasons: the maximum flash size is 16MB (24 bits), and each block is 32 bytes so the least significant 5 bits are always zero.
- There is a particular mapping from each of the 19 block offset bits to the 256 bits of the flash encryption key to determine which bit is XORed with which. See the variable `_FLASH_ENCRYPTION_TWEAK_PATTERN` in the `espsecure.py` source code for complete mapping.
- To see the full flash encryption algorithm implemented in Python, refer to the `_flash_encryption_operation()` function in the `espsecure.py` source code.

4.14 Hardware Abstraction

ESP-IDF provides a group of APIs for hardware abstraction. These APIs allow you to control peripherals at different levels of abstraction, giving you more flexibility compared to using only the ESP-IDF drivers to interact with hardware. ESP-IDF Hardware abstraction is likely to be useful for writing high-performance bare-metal drivers, or for attempting to port an ESP chip to another platform.

This guide is split into the following sections:

1. *Architecture*
2. *LL (Low Level) Layer*
3. *HAL (Hardware Abstraction Layer)*

Warning: Hardware abstraction API (excluding the driver and `xxx_types.h`) should be considered an experimental feature, thus cannot be considered public API. The hardware abstraction API does not adhere to the API name changing restrictions of ESP-IDF's versioning scheme. In other words, it is possible that Hardware Abstraction API may change in between non-major release versions.

Note: Although this document mainly focuses on hardware abstraction of peripherals, e.g., UART, SPI, I2C, certain layers of hardware abstraction extend to other aspects of hardware as well, e.g., some of the CPU's features are partially abstracted.

4.14.1 Architecture

Hardware abstraction in ESP-IDF is comprised of the following layers, ordered from low level of abstraction that is closer to hardware, to high level of abstraction that is further away from hardware.

- Low Level (LL) Layer
- Hardware Abstraction Layer (HAL)
- Driver Layers

The LL Layer, and HAL are entirely contained within the `hal` component. Each layer is dependent on the layer below it, i.e, driver depends on HAL, HAL depends on LL, LL depends on the register header files.

For a particular peripheral `xxx`, its hardware abstraction generally consists of the header files described in the table below. Files that are **Target Specific** have a separate implementation for each target, i.e., a separate copy for each chip. However, the `#include` directive is still target-independent, i.e., is the same for different targets, as the build system automatically includes the correct version of the header and source files.

Table 21: Hardware Abstraction Header Files

Include Directive	Target Specific	Description
`#include 'soc/xxx_caps.h"`	Y	This header contains a list of C macros specifying the various capabilities of the ESP32's peripheral `xxx`. Hardware capabilities of a peripheral include things such as the number of channels, DMA support, hardware FIFO/buffer lengths, etc.
`#include "soc/xxx_struct.h"` `#include "soc/xxx_reg.h"`	Y	The two headers contain a representation of a peripheral's registers in C structure and C macro format respectively, allowing you to operate a peripheral at the register level via either of these two header files.
`#include "soc/xxx_pins.h"`	Y	If certain signals of a peripheral are mapped to a particular pin of the ESP32, their mappings are defined in this header as C macros.
`#include "soc/xxx_periph.h"`	N	This header is mainly used as a convenience header file to automatically include `xxx_caps.h`, `xxx_struct.h`, and `xxx_reg.h`.
`#include "hal/xxx_types.h`	N	This header contains type definitions and macros that are shared among the LL, HAL, and driver layers. Moreover, it is considered public API thus can be included by the application level. The shared types and definitions usually related to non-implementation specific concepts such as the following: • Protocol-related types/macros such a frames, modes, common bus speeds, etc. • Features/characteristics of an `xxx` peripheral that are likely to be present on any implementation (implementation-independent) such as channels, operating modes, signal amplification or attenuation intensities, etc.
`#include "hal/xxx_ll.h"`	Y	This header contains the Low Level (LL) Layer of hardware abstraction. LL Layer API are primarily used to abstract away register operations into readable functions.
`#include "hal/xxx_hal.h"`	Y	The Hardware Abstraction Layer (HAL) is used to abstract away peripheral operation steps into functions (e.g., reading a buffer, starting a transmission, handling an event, etc). The HAL is built on top of the LL Layer.
`#include "driver/xxx.h"`	N	The driver layer is the highest level of ESP-IDF's hardware abstraction. Driver layer API are meant to be called from ESP-IDF applications, and internally utilize OS primitives. Thus, driver layer API are event-driven, and can used in a multi-threaded environment.

4.14.2 LL (Low Level) Layer

The primary purpose of the LL Layer is to abstract away register field access into more easily understandable functions. LL functions essentially translate various in/out arguments into the register fields of a peripheral in the form of get/set functions. All the necessary bit shifting, masking, offsetting, and endianness of the register fields should be handled by the LL functions.

```
//Inside xxx_ll.h

static inline void xxx_ll_set_baud_rate(xxx_dev_t *hw,
                                        xxx_ll_clk_src_t clock_source,
                                        uint32_t baud_rate) {
    uint32_t src_clk_freq = (source_clk == XXX_SCLK_APB) ? APB_CLK_FREQ : REF_CLK_
↪FREQ;
```

(continues on next page)

(continued from previous page)

```
    uint32_t clock_divider = src_clk_freq / baud;
    // Set clock select field
    hw->clk_div_reg.divider = clock_divider >> 4;
    // Set clock divider field
    hw->config.clk_sel = (source_clk == XXX_SCLK_APB) ? 0 : 1;
}

static inline uint32_t xxx_ll_get_rx_byte_count(xxx_dev_t *hw) {
    return hw->status_reg.rx_cnt;
}
```

The code snippet above illustrates typical LL functions for a peripheral `xxx`. LL functions typically have the following characteristics:

- All LL functions are defined as `static inline` so that there is minimal overhead when calling these functions due to compiler optimization. These functions are not guaranteed to be inlined by the compiler, so any LL function that is called when the cache is disabled (e.g., from an IRAM ISR context) should be marked with `__attribute__((always_inline))`.
- The first argument should be a pointer to a `xxx_dev_t` type. The `xxx_dev_t` type is a structure representing the peripheral's registers, thus the first argument is always a pointer to the starting address of the peripheral's registers. Note that in some cases where the peripheral has multiple channels with identical register layouts, `xxx_dev_t *hw` may point to the registers of a particular channel instead.
- LL functions should be short, and in most cases are deterministic. In other words, in the worst case, runtime of the LL function can be determined at compile time. Thus, any loops in LL functions should be finite bounded; however, there are currently a few exceptions to this rule.
- LL functions are not thread-safe, it is the responsibility of the upper layers (driver layer) to ensure that registers or register fields are not accessed concurrently.

4.14.3 HAL (Hardware Abstraction Layer)

The HAL layer models the operational process of a peripheral as a set of general steps, where each step has an associated function. For each step, the details of a peripheral's register implementation (i.e., which registers need to be set/read) are hidden (abstracted away) by the HAL. By modeling peripheral operation as a set of functional steps, any minor hardware implementation differences of the peripheral between different targets or chip versions can be abstracted away by the HAL (i.e., handled transparently). In other words, the HAL API for a particular peripheral remains mostly the same across multiple targets/chip versions.

The following HAL function examples are selected from the Watchdog Timer HAL as each function maps to one of the steps in a WDT's operation life cycle, thus illustrating how a HAL abstracts a peripheral's operation into functional steps.

```
// Initialize one of the WDTs
void wdt_hal_init(wdt_hal_context_t *hal, wdt_inst_t wdt_inst, uint32_t prescaler,␣
↪bool enable_intr);

// Configure a particular timeout stage of the WDT
void wdt_hal_config_stage(wdt_hal_context_t *hal, wdt_stage_t stage, uint32_t␣
↪timeout, wdt_stage_action_t behavior);

// Start the WDT
void wdt_hal_enable(wdt_hal_context_t *hal);

// Feed (i.e., reset) the WDT
void wdt_hal_feed(wdt_hal_context_t *hal);

// Handle a WDT timeout
void wdt_hal_handle_intr(wdt_hal_context_t *hal);

// Stop the WDT
```

(continues on next page)

(continued from previous page)

```
void wdt_hal_disable(wdt_hal_context_t *hal);

// De-initialize the WDT
void wdt_hal_deinit(wdt_hal_context_t *hal);
```

HAL functions generally have the following characteristics:

- The first argument to a HAL function has the `xxx_hal_context_t *` type. The HAL context type is used to store information about a particular instance of the peripheral (i.e., the context instance). A HAL context is initialized by the `xxx_hal_init()` function and can store information such as the following:
  - The channel number of this instance
  - Pointer to the peripheral's (or channel's) registers (i.e., a `xxx_dev_t *` type)
  - Information about an ongoing transaction (e.g., pointer to DMA descriptor list in use)
  - Some configuration values for the instance (e.g., channel configurations)
  - Variables to maintain state information regarding the instance (e.g., a flag to indicate if the instance is waiting for transaction to complete)
- HAL functions should not contain any OS primitives such as queues, semaphores, mutexes, etc. All synchronization/concurrency should be handled at higher layers (e.g., the driver).
- Some peripherals may have steps that cannot be further abstracted by the HAL, thus end up being a direct wrapper (or macro) for an LL function.
- Some HAL functions may be placed in IRAM thus may carry an `IRAM_ATTR` or be placed in a separate `xxx_hal_iram.c` source file.

4.15 High-Level Interrupts

The Xtensa architecture has support for 32 interrupts, divided over 7 levels (levels 1 to 7, with 7 being an NMI), plus an assortment of exceptions. On the ESP32, the interrupt mux allows most interrupt sources to be routed to these interrupts using the *interrupt allocator*. Normally, interrupts will be written in C, but ESP-IDF allows high-level interrupts to be written in assembly as well, resulting in very low interrupt latencies.

4.15.1 Interrupt Levels

Level	Symbol	Remark
1	N/A	Exception and level 0 interrupts. Handled by ESP-IDF
2-3	N/A	Medium level interrupts. Handled by ESP-IDF
4	xt_highint4	Free to use *(See 1)*
5	xt_highint5	Normally used by ESP-IDF debug logic *(See 1)*
NMI	xt_nmi	Free to use
dbg	xt_debugexception	Debug exception. Called on e.g. a BREAK instruction. *(See 2)*

The following notes give more information about the items in the tables above.

1. ESP-IDF debug logic can be configured to run on *xt_highint4* or *xt_highint5* in *CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL*. Bluetooth's interrupt can be configured to run on level 4 by enabling *CONFIG_BTDM_CTRL_HLI*. If *CONFIG_BTDM_CTRL_HLI* is enabled, ESP-IDF debug logic must be running on level 5 interrupt.
2. If *CONFIG_BTDM_CTRL_HLI* is enabled, *xt_debugexception* is used to fix live lock issue in ESP32 ECO3.

Using these symbols is done by creating an assembly file (suffix .S) and defining the named symbols, like this:

```
    .section .iram1,"ax"
    .global     xt_highint5
    .type       xt_highint5,@function
    .align      4
xt_highint5:
```

(continues on next page)

(continued from previous page)

```
    ... your code here
    rsr     a0, EXCSAVE_5
    rfi     5
```

For a real-life example, see the esp_system/port/soc/esp32/highint_hdl.S file; the panic handler interrupt is implemented there.

4.15.2 Notes

- Do not call C code from a high-level interrupt; as these interrupts are run from a critical section, this can cause the target to crash. Note that although the panic handler interrupt does call normal C code, this exception is allowed due to the fact that this handler never returns (i.e., the application will not continue to run after the panic handler). so breaking C code execution flow is not a problem.
  When *CONFIG_BTDM_CTRL_HLI* is enabled, C code is also called from a high-level interrupt, this is possible thanks to some additional protection added to it.
- Make sure your assembly code gets linked in. Indeed, as the free-to-use symbols are declared as weak, the linker may discard the file containing the symbol. This will happen if the only symbol defined, or used, from the user file is the `xt_*` free-to-use symbol. To avoid this, in the assembly file containing the `xt_*` symbol, define another symbol, like:

```
        .global ld_include_my_isr_file
   ld_include_my_isr_file:

Here it is called ``ld_include_my_isr_file`` but can have any name, as long as␣
↪it is not defined anywhere else in the project.

Then, in the component ``CMakeLists.txt``, add this name as an unresolved␣
↪symbol to the ld command line arguments::

    target_link_libraries(${COMPONENT_TARGET} "-u ld_include_my_isr_file")

This should cause the linker to always include the file defining ``ld_include_
↪my_isr_file``, causing the ISR to always be linked in.
```

- High-level interrupts can be routed and handled using *esp_intr_alloc()* and associated functions. The handler and handler arguments to *esp_intr_alloc()* must be NULL, however.
- In theory, medium priority interrupts could also be handled in this way. ESP-IDF does not support this yet.

4.16 JTAG Debugging

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB. The document is structured as follows:

Introduction Introduction to the purpose of this guide.
How it Works? Description how ESP32, JTAG interface, OpenOCD and GDB are interconnected and working together to enable debugging of ESP32.
Selecting JTAG Adapter What are the criteria and options to select JTAG adapter hardware.
Setup of OpenOCD Procedure to install OpenOCD and verify that it is installed.
Configuring ESP32 Target Configuration of OpenOCD software and setting up of JTAG adapter hardware, which together make up the debugging target.
Launching Debugger Steps to start up a debug session with GDB from *Eclipse* and from *Command Line*.
Debugging Examples If you are not familiar with GDB, check this section for debugging examples provided from *Eclipse* as well as from *Command Line*.
Building OpenOCD from Sources Procedure to build OpenOCD from sources for *Windows*, *Linux* and *macOS* operating systems.

Tips and Quirks This section provides collection of tips and quirks related to JTAG debugging of ESP32 with OpenOCD and GDB.

4.16.1 Introduction

The ESP32 has two powerful Xtensa cores, allowing for a great deal of variety of program architectures. The FreeRTOS OS that comes with ESP-IDF is capable of multi-core preemptive scheduling, allowing for an intuitive way of writing software.

The downside of the ease of programming is that debugging without the right tools is harder: figuring out a bug that is caused by two threads, running even simultaneously on two different CPU cores, can take a long time when all you have are `printf()` statements. A better (and in many cases quicker) way to debug such problems is by using a debugger, connected to the processors over a debug port.

Espressif has ported OpenOCD to support the ESP32 processor and the multi-core FreeRTOS（which is the foundation of most ESP32 apps). Additionally, some extra tools have been written to provide extra features that OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB under Linux, Windows and macOS. Except for OS specific installation procedures, the s/w user interface and use procedures are the same across all supported operating systems.

Note: Screenshots presented in this document have been made for Eclipse Neon 3 running on Ubuntu 16.04 LTS. There may be some small differences in what a particular user interface looks like, depending on whether you are using Windows, macOS or Linux and / or a different release of Eclipse.

4.16.2 How it Works?

The key software and hardware components that perform debugging of ESP32 with OpenOCD over JTAG (Joint Test Action Group) interface is presented in the diagram below under the "Debugging With JTAG" label. These components include xtensa-esp32-elf-gdb debugger, OpenOCD on chip debugger, and the JTAG adapter connected to ESP32 target.

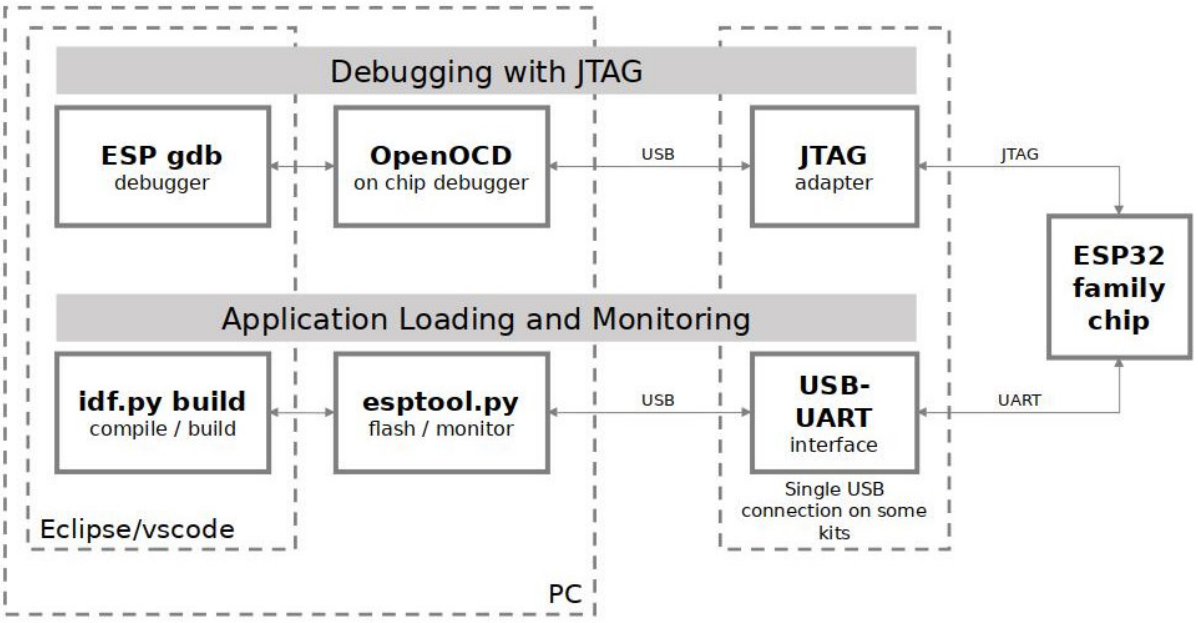


Fig. 36: JTAG debugging - overview diagram

Likewise, the "Application Loading and Monitoring" label indicates the key software and hardware components that allow an application to be compiled, built, and flashed to ESP32, as well as to provide means to monitor diagnostic messages from ESP32.

"Debugging With JTAG" and "Application Loading and Monitoring" is integrated under the Eclipse IDE in order to provide a quick and easy transition between writing/compiling/loading/debugging code. The Eclipse IDE (and the integrated debugging software) is available for Windows, Linux and macOS platforms. Depending on user preferences, both the debugger and `idf.py build` can also be used directly from terminal/command line, instead of Eclipse.

If the *ESP-WROVER-KIT* is used, then connection from PC to ESP32 is done effectively with a single USB cable. This is made possible by the FT2232H chip, which provides two USB channels, one for JTAG and the other for UART connection.

4.16.3 Selecting JTAG Adapter

The quickest and most convenient way to start with JTAG debugging is by using *ESP-WROVER-KIT*. Each version of this development board has JTAG interface already built in. No need for an external JTAG adapter and extra wiring / cable to connect JTAG to ESP32. ESP-WROVER-KIT is using FT2232H JTAG interface operating at 20 MHz clock speed, which is difficult to achieve with an external adapter.

If you decide to use separate JTAG adapter, look for one that is compatible with both the voltage levels on the ESP32 as well as with the OpenOCD software. The JTAG port on the ESP32 is an industry-standard JTAG port which lacks (and does not need) the TRST pin. The JTAG I/O pins all are powered from the VDD_3P3_RTC pin (which normally would be powered by a 3.3 V rail) so the JTAG adapter needs to be able to work with JTAG pins in that voltage range.

On the software side, OpenOCD supports a fair amount of JTAG adapters. See https://openocd.org/doc/html/Debug-Adapter-Hardware.html for an (unfortunately slightly incomplete) list of the adapters OpenOCD works with. This page lists SWD-compatible adapters as well; take note that the ESP32 does not support SWD. JTAG adapters that are hardcoded to a specific product line, e.g. ST-LINK debugging adapters for STM32 families, will not work.

The minimal signalling to get a working JTAG connection are TDI, TDO, TCK, TMS and GND. Some JTAG debuggers also need a connection from the ESP32 power line to a line called e.g. Vtar to set the working voltage. SRST can optionally be connected to the CH_PD of the ESP32, although for now, support in OpenOCD for that line is pretty minimal.

ESP-Prog is an example for using an external board for debugging by connecting it to the JTAG pins of ESP32.

4.16.4 Setup of OpenOCD

If you have already set up ESP-IDF with CMake build system according to the *Getting Started Guide*, then OpenOCD is already installed. After *setting up the environment* in your terminal, you should be able to run OpenOCD. Check this by executing the following command:

```
openocd --version
```

The output should be as follows (although the version may be more recent than listed here):

```
Open On-Chip Debugger  v0.10.0-esp32-20190708 (2019-07-08-11:04)
Licensed under GNU GPL v2
For bug reports, read
    https://openocd.org/doc/doxygen/bugs.html
```

You may also verify that OpenOCD knows where its configuration scripts are located by printing the value of `OPENOCD_SCRIPTS` environment variable, by typing `echo $OPENOCD_SCRIPTS` (for Linux and macOS) or `echo %OPENOCD_SCRIPTS%` (for Windows). If a valid path is printed, then OpenOCD is set up correctly.

If any of these steps do not work, please go back to the *setting up the tools* section of the Getting Started Guide.

Note: It is also possible to build OpenOCD from source. Please refer to *Building OpenOCD from Sources* section for details.

4.16.5 Configuring ESP32 Target

Once OpenOCD is installed, you can proceed to configuring the ESP32 target (i.e ESP32 board with JTAG interface). Configuring the target is split into the following three steps:

- *Configure and connect JTAG interface*
- *Run OpenOCD*
- *Upload application for debugging*

Configure and connect JTAG interface

This step depends on the JTAG and ESP32 board you are using (see the two cases described below).

Configure ESP-WROVER-KIT JTAG Interface

All versions of ESP-WROVER-KIT boards have built-in JTAG functionality. Putting it to work requires setting jumpers or DIP switches to enable JTAG functionality, and configuring USB drivers. Please refer to step by step instructions below.

Configure Hardware

- Enable on-board JTAG functionality by setting JP8 according to *ESP-WROVER-KIT V4.1 Getting Started Guide*, Section *Setup Options*.
- Verify if ESP32 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation:

Table 22: ESP32 JTAG pins

ESP32 Pin	JTAG Signal
MTDO / GPIO15	TDO
MTDI / GPIO12	TDI
MTCK / GPIO13	TCK
MTMS / GPIO14	TMS

Configure USB Drivers Install and configure USB drivers, so OpenOCD is able to communicate with JTAG interface on ESP-WROVER-KIT board as well as with UART interface used to upload application for flash. Follow steps below specific to your operating system.

Note: ESP-WROVER-KIT uses an FT2232 adapter. The following instructions can also be used for other FT2232 based JTAG adapters.

Windows

1. Using standard USB A / micro USB B cable connect ESP-WROVER-KIT to the computer. Switch the ESP-WROVER-KIT on.
2. Wait until USB ports of ESP-WROVER-KIT are recognized by Windows and drives are installed. If they do not install automatically, then download them from https://ftdichip.com/drivers/d2xx-drivers/ and install manually.
3. Download Zadig tool (Zadig_X.X.exe) from https://zadig.akeo.ie/ and run it.
4. In Zadig tool go to "Options" and check "List All Devices".

5. Check the list of devices that should contain two ESP-WROVER-KIT specific USB entries: "Dual RS232-HS (Interface 0)" and "Dual RS232-HS (Interface 1)". The driver name would be "FTDIBUS (vxxxx)" and USB ID: 0403 6010.

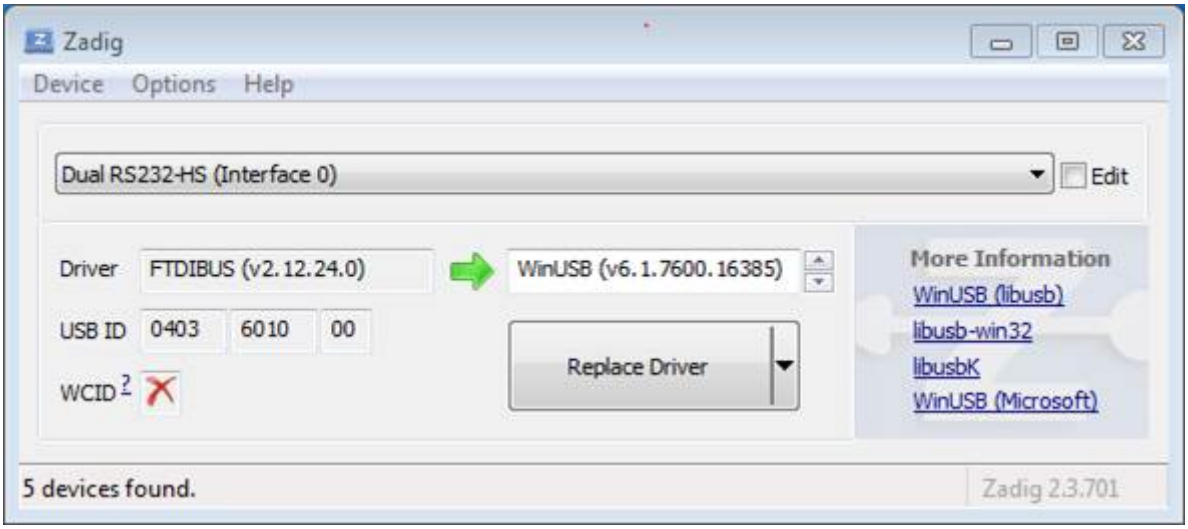


Fig. 37: Configuration of JTAG USB driver in Zadig tool

6. The first device (Dual RS232-HS (Interface 0)) is connected to the JTAG port of the ESP32. Original "FTDIBUS (vxxxx)" driver of this device should be replaced with "WinUSB (v6xxxxx)". To do so, select "Dual RS232-HS (Interface 0) and reinstall attached driver to the "WinUSB (v6xxxxx)", see picture above.

Note: Do not change the second device "Dual RS232-HS (Interface 1)". It is routed to ESP32's serial port (UART) used for upload of application to ESP32's flash.

Now ESP-WROVER-KIT's JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

Linux

1. Using standard USB A / micro USB B cable connect ESP-WROVER-KIT board to the computer. Power on the board.
2. Open a terminal, enter `ls -l /dev/ttyUSB*` command and check, if board's USB ports are recognized by the OS. You are looking for similar result:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jul 10 19:04 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 Jul 10 19:04 /dev/ttyUSB1
```

3. Following section "Permissions delegation" in OpenOCD's README, set up the access permissions to both USB ports.
4. Log off and login, then cycle the power to the board to make the changes effective. In terminal enter again `ls -l /dev/ttyUSB*` command to verify, if group-owner has changed from `dialout` to `plugdev`:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw-r-- 1 root plugdev 188, 0 Jul 10 19:07 /dev/ttyUSB0
crw-rw-r-- 1 root plugdev 188, 1 Jul 10 19:07 /dev/ttyUSB1
```

   If you see similar result and you are a member of `plugdev` group, then the set up is complete.
   The `/dev/ttyUSBn` interface with lower number is used for JTAG communication. The other interface is routed to ESP32's serial port (UART) used for upload of application to ESP32's flash.

Now ESP-WROVER-KIT's JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

MacOS On macOS, using FT2232 for JTAG and serial port at the same time needs some additional steps. When the OS loads FTDI serial port driver, it does so for both channels of FT2232 chip. However only one of these channels is used as a serial port, while the other is used as JTAG. If the OS has loaded FTDI serial port driver for the channel used for JTAG, OpenOCD will not be able to connect to the chip. There are two ways around this:

1. Manually unload the FTDI serial port driver before starting OpenOCD, start OpenOCD, then load the serial port driver.
2. Modify FTDI driver configuration so that it doesn't load itself for channel B of FT2232 chip, which is the channel used for JTAG on ESP-WROVER-KIT.

Manually unloading the driver

1. Install FTDI driver from https://ftdichip.com/drivers/vcp-drivers/
2. Connect USB cable to the ESP-WROVER-KIT.
3. Unload the serial port driver:

   ```
   sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
   ```

   In some cases you may need to unload Apple's FTDI driver as well:
   - macOS < 10.15:

     ```
     sudo kextunload -b com.apple.driver.AppleUSBFTDI
     ```

   - macOS 10.15:

     ```
     sudo kextunload -b com.apple.DriverKit-AppleUSBFTDI
     ```

   Warning: Attempting to use serial over the wrong channel with the FTDI driver will cause a kernel panic. The ESP-WROVER-KIT uses channel A for JTAG and channel B for serial.

4. Run OpenOCD:

   ```
   openocd -f board/esp32-wrover-kit-3.3v.cfg
   ```

5. In another terminal window, load FTDI serial port driver again:

   ```
   sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
   ```

Note: If you need to restart OpenOCD, there is no need to unload FTDI driver again —just stop OpenOCD and start it again. The driver only needs to be unloaded if ESP-WROVER-KIT was reconnected or power was toggled.

This procedure can be wrapped into a shell script, if desired.

Modifying FTDI driver In a nutshell, this approach requires modification to FTDI driver configuration file, which prevents the driver from being loaded for channel B of FT2232H.

Note: Other boards may use channel A for JTAG, so use this option with caution.

Warning: This approach also needs signature verification of drivers to be disabled, so may not be acceptable for all users.

1. Open FTDI driver configuration file using a text editor (note `sudo`):

   ```
   sudo nano /Library/Extensions/FTDIUSBSerialDriver.kext/Contents/Info.plist
   ```

2. Find and delete the following lines:

```
<key>FT2232H_B</key>
<dict>
    <key>CFBundleIdentifier</key>
    <string>com.FTDI.driver.FTDIUSBSerialDriver</string>
    <key>IOClass</key>
    <string>FTDIUSBSerialDriver</string>
    <key>IOProviderClass</key>
    <string>IOUSBInterface</string>
    <key>bConfigurationValue</key>
    <integer>1</integer>
    <key>bInterfaceNumber</key>
    <integer>1</integer>
    <key>bcdDevice</key>
    <integer>1792</integer>
    <key>idProduct</key>
    <integer>24592</integer>
    <key>idVendor</key>
    <integer>1027</integer>
</dict>
```

3. Save and close the file
4. Disable driver signature verification:
    1. Open Apple logo menu, choose "Restart…"
    2. When you hear the chime after reboot, press CMD+R immediately
    3. Once Recovery mode starts up, open Terminal
    4. Run the command:

    ```
    csrutil enable --without kext
    ```

    5. Restart again

After these steps, serial port and JTAG can be used at the same time.

To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

Configure Other JTAG Interfaces

For guidance about which JTAG interface to select when using OpenOCD with ESP32, refer to the section *Selecting JTAG Adapter*. Then follow the configuration steps below to get it working.

Configure Hardware

1. Identify all pins/signals on JTAG interface and ESP32 board that should be connected to establish communication.

Table 23: ESP32 JTAG pins

ESP32 Pin	JTAG Signal
MTDO / GPIO15	TDO
MTDI / GPIO12	TDI
MTCK / GPIO13	TCK
MTMS / GPIO14	TMS

2. Verify if ESP32 pins used for JTAG communication are not connected to some other hardware that may disturb JTAG operation.
3. Connect identified pin/signals of ESP32 and JTAG interface.

Configure Drivers You may need to install driver software to make JTAG work with computer. Refer to documentation of your JTAG adapter for related details.

Connect Connect JTAG interface to the computer. Power on ESP32 and JTAG interface boards. Check if the JTAG interface is visible on the computer.

To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

Run OpenOCD

Once target is configured and connected to computer, you are ready to launch OpenOCD.

Open a terminal and set it up for using the ESP-IDF as described in the *setting up the environment* section of the Getting Started Guide. Then run OpenOCD (this command works on Windows, Linux, and macOS):

```
openocd -f board/esp32-wrover-kit-3.3v.cfg
```

Note: The files provided after `-f` above are specific for ESP-WROVER-KIT with ESP32-WROOM-32 module. You may need to provide different files depending on the hardware that is used. For guidance see *Configuration of OpenOCD for specific target*.

For example, `board/esp32c3-ftdi.cfg` can be used for a custom board with an FT2232H or FT232H chip used for JTAG connection, or with ESP-Prog.

You should now see similar output (this log is for ESP-WROVER-KIT with ESP32-WROOM-32 module):

```
user-name@computer-name:~/esp/esp-idf$ openocd -f board/esp32-wrover-kit-3.3v.cfg
Open On-Chip Debugger  v0.10.0-esp32-20190708 (2019-07-08-11:04)
Licensed under GNU GPL v2
For bug reports, read
        https://openocd.org/doc/doxygen/bugs.html
none separate
adapter speed: 20000 kHz
force hard breakpoints
Info : ftdi: if you experience problems at higher adapter clocks, try the command
↪"ftdi_tdo_sample_edge falling"
Info : clock speed 20000 kHz
Info : JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),␣
↪part: 0x2003, ver: 0x1)
Info : JTAG tap: esp32.cpu1 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),␣
↪part: 0x2003, ver: 0x1)
Info : esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
Info : esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
```

- If there is an error indicating permission problems, please see section on "Permissions delegation" in the OpenOCD README file located in the `~/esp/openocd-esp32` directory.
- In case there is an error in finding the configuration files, e.g. `Can't find board/esp32-wrover-kit-3.3v.cfg`, check if the `OPENOCD_SCRIPTS` environment variable is set correctly. This variable is used by OpenOCD to look for the files specified after the `-f` option. See *Setup of OpenOCD* section for details. Also check if the file is indeed under the provided path.
- If you see JTAG errors (e.g., `...all ones` or `...all zeroes`), please check your JTAG connections, whether other signals are connected to JTAG besides ESP32's pins, and see if everything is powered on correctly.

Upload application for debugging

Build and upload your application to ESP32 as usual, see *Step 5. First Steps on ESP-IDF*.

Another option is to write application image to flash using OpenOCD via JTAG with commands like this:

```
openocd -f board/esp32-wrover-kit-3.3v.cfg -c "program_esp filename.bin 0x10000␣
↪verify exit"
```

OpenOCD flashing command `program_esp` has the following format:

```
program_esp <image_file> <offset> [verify] [reset] [exit]
```

- `image_file` - Path to program image file.
- `offset` - Offset in flash bank to write image.
- `verify` - Optional. Verify flash contents after writing.
- `reset` - Optional. Reset target after programing.
- `exit` - Optional. Finally exit OpenOCD.

You are now ready to start application debugging. Follow the steps described in the section below.

4.16.6 Launching Debugger

The toolchain for ESP32 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename: xtensa-esp32-elf-gdb. GDB can be called and operated directly from command line in a terminal. Another option is to call it from within IDE (like Eclipse, Visual Studio Code, etc.) and operate indirectly with help of GUI instead of typing commands in a terminal.

Both options of using debugger are discussed under links below.

- *Eclipse*
- *Command Line*

It is recommended to first check if debugger works from *Command Line* and then move to using *Eclipse*.

4.16.7 Debugging Examples

This section is intended for users not familiar with GDB. It presents example debugging session from *Eclipse* using simple application available under get-started/blink and covers the following debugging actions:

1. *Navigating through the code, call stack and threads*
2. *Setting and clearing breakpoints*
3. *Halting the target manually*
4. *Stepping through the code*
5. *Checking and setting memory*
6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Similar debugging actions are provided using GDB from *Command Line*.

Before proceeding to examples, set up your ESP32 target and load it with get-started/blink.

4.16.8 Building OpenOCD from Sources

Please refer to separate documents listed below, that describe build process.

Building OpenOCD from Sources for Windows

Note: This document outlines how to build a binary of OpenOCD from its source files instead of downloading the pre-built binary. For a quick setup, users can download a pre-built binary of OpenOCD from Espressif GitHub instead of compiling it themselves (see *Setup of OpenOCD* for more details).

Note: All code snippets in this document are assumed to be running in an MSYS2 shell with the MINGW32 subsystem.

Install Dependencies Install packages that are required to compile OpenOCD:

```
pacman -S --noconfirm --needed autoconf automake git make \
mingw-w64-i686-gcc \
mingw-w64-i686-toolchain \
mingw-w64-i686-libtool \
mingw-w64-i686-pkg-config \
mingw-w64-cross-winpthreads-git \
p7zip
```

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif's GitHub under https://github.com/espressif/openocd-esp32. These source files can be pulled via Git using the following commands:

```
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Downloading libusb The libusb library is also required when building OpenOCD. The following commands will download a particular release of libusb and uncompress it to the current directory.

```
wget https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.7z
7z x -olibusb ./libusb-1.0.22.7z
```

We now need to export the following variables such that the libusb library gets linked into the OpenOCD build.

```
export CPPFLAGS="$CPPFLAGS -I${PWD}/libusb/include/libusb-1.0"
export LDFLAGS="$LDFLAGS -L${PWD}/libusb/MinGW32/.libs/dll"
```

Build OpenOCD The following commands will configure OpenOCD then build it.

```
cd ~/esp/openocd-esp32
export CPPFLAGS="$CPPFLAGS -D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="
↪$CFLAGS -Wno-error"
./bootstrap
./configure --disable-doxygen-pdf --enable-ftdi --enable-jlink --enable-ulink --
↪build=i686-w64-mingw32 --host=i686-w64-mingw32
make
cp ../libusb/MinGW32/dll/libusb-1.0.dll ./src
cp /opt/i686-w64-mingw32/bin/libwinpthread-1.dll ./src
```

Once the build is completed, the OpenOCD binary will be placed in `~/esp/openocd-esp32/src/`.

You can then optionally call `make install`. This will copy the OpenOCD binary to a user specified location.

- This location can be specified when OpenOCD is configured, or by setting `export DESTDIR="/custom/install/dir"` before calling `make install`.
- If you have an existing OpenOCD (from e.g. another development platform), you may want to skip this call as your existing OpenOCD may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under `OpenOCD configuration summary`.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
- For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.Windows`.
- Don't forget to copy *libusb-1.0.dll* and *libwinpthread-1.dll* into *OOCD_INSTALLDIR/bin* from `~/esp/openocd-esp32/src`.

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src` directory.

Full Listing For greater convenience, all of commands called throughout the OpenOCD build process have been listed in the code snippet below. Users can copy this code snippet into a shell script then execute it:

```
pacman -S --noconfirm --needed autoconf automake git make mingw-w64-i686-gcc mingw-
↪w64-i686-toolchain mingw-w64-i686-libtool mingw-w64-i686-pkg-config mingw-w64-
↪cross-winpthreads-git p7zip
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git

wget https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.7z
7z x -olibusb ./libusb-1.0.22.7z
export CPPFLAGS="$CPPFLAGS -I${PWD}/libusb/include/libusb-1.0"; export LDFLAGS="
↪$LDFLAGS -L${PWD}/libusb/MinGW32/.libs/dll"

export CPPFLAGS="$CPPFLAGS -D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="
↪$CFLAGS -Wno-error"
cd ~/esp/openocd-esp32
./bootstrap
./configure --disable-doxygen-pdf --enable-ftdi --enable-jlink --enable-ulink --
↪build=i686-w64-mingw32 --host=i686-w64-mingw32
make
cp ../libusb/MinGW32/dll/libusb-1.0.dll ./src
cp /opt/i686-w64-mingw32/bin/libwinpthread-1.dll ./src

# # optional
# export DESTDIR="$PWD"
# make install
# cp ./src/libusb-1.0.dll $DESTDIR/mingw32/bin
# cp ./src/libwinpthread-1.dll $DESTDIR/mingw32/bin
```

Next Steps To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

Building OpenOCD from Sources for Linux

The following instructions are alternative to downloading binary OpenOCD from Espressif GitHub. To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section *Setup of OpenOCD*.

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under https://github.com/espressif/openocd-esp32. To download the sources, use the following commands:

```
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies Install packages that are required to compile OpenOCD.

Note: Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

```
sudo apt-get install make
sudo apt-get install libtool
sudo apt-get install pkg-config
sudo apt-get install autoconf
sudo apt-get install automake
sudo apt-get install texinfo
sudo apt-get install libusb-1.0
```

Note:

- Version of pkg-config should be 0.2.3 or above.
- Version of autoconf should be 2.6.4 or above.
- Version of automake should be 1.9 or above.
- When using USB-Blaster, ASIX Presto, OpenJTAG and FT2232 as adapters, drivers libFTDI and FTD2XX need to be downloaded and installed.
- When using CMSIS-DAP, HIDAPI is needed.

Build OpenOCD Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under `OpenOCD configuration summary`.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
- For details concerning compiling OpenOCD, please refer to `openocd-esp32/README`.

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/openocd-esp32/bin` directory.

Next Steps To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

Building OpenOCD from Sources for MacOS

The following instructions are alternative to downloading binary OpenOCD from Espressif GitHub. To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section *Setup of OpenOCD*.

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under https://github.com/espressif/openocd-esp32. To download the sources, use the following commands:

```
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies Install packages that are required to compile OpenOCD using Homebrew:

```
brew install automake libtool libusb wget gcc@4.9 pkg-config
```

Build OpenOCD Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under `OpenOCD configuration summary`.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
- For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.OSX`.

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src/openocd` directory.

Next Steps To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

The examples of invoking OpenOCD in this document assume using pre-built binary distribution described in section *Setup of OpenOCD*.

To use binaries build locally from sources, change the path to OpenOCD executable to `src/openocd` and set the `OPENOCD_SCRIPTS` environment variable so that OpenOCD can find the configuration files. For Linux and macOS:

```
cd ~/esp/openocd-esp32
export OPENOCD_SCRIPTS=$PWD/tcl
```

For Windows:

```
cd %USERPROFILE%\esp\openocd-esp32
set "OPENOCD_SCRIPTS=%CD%\tcl"
```

Example of invoking OpenOCD build locally from sources, for Linux and macOS:

```
src/openocd -f board/esp32-wrover-kit-3.3v.cfg
```

and Windows:

```
src\openocd -f board\esp32-wrover-kit-3.3v.cfg
```

4.16.9 Tips and Quirks

This section provides collection of links to all tips and quirks referred to from various parts of this guide.

Tips and Quirks

This section provides collection of all tips and quirks referred to from various parts of this guide.

Breakpoints and watchpoints available ESP32 debugger supports 2 hardware implemented breakpoints and 64 software ones. Hardware breakpoints are implemented by ESP32 chip's logic and can be set anywhere in the code: either in flash or IRAM program's regions. Additionally there are 2 types of software breakpoints implemented by OpenOCD: flash (up to 32) and IRAM (up to 32) breakpoints. Currently GDB can not set software breakpoints in flash. So until this limitation is removed those breakpoints have to be emulated by OpenOCD as hardware ones (see *below* for details). ESP32 also supports 2 watchpoints, so 2 variables can be watched for change or read by the GDB command `watch myVariable`. Note that menuconfig option *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK* uses the last watchpoint and will not provide expected results, if you also try to use it within OpenOCD / GDB. See menuconfig's help for detailed description.

What else should I know about breakpoints? Emulating part of hardware breakpoints using software flash ones means that the GDB command `hb myFunction` which is invoked for function in flash will use pure hardware breakpoint if it is avalable otherwise one of the 32 software flash breakpoints is used. The same rule applies to `b myFunction`-like commands. In this case GDB will decide what type of breakpoint to set itself. If `myFunction` is resided in writable region (IRAM) software IRAM breakpoint will be used otherwise hardware or software flash breakpoint is used as it is done for `hb` command.

Flash Mappings vs SW Flash Breakpoints In order to set/clear software breakpoints in flash, OpenOCD needs to know their flash addresses. To accomplish conversion from the ESP32 address space to the flash one, OpenOCD uses mappings of program's code regions resided in flash. Those mappings are kept in the image header which is prepended to program binary data (code and data segments) and is specific to every application image written to the flash. So to support software flash breakpoints OpenOCD should know where application image under debugging is resided in the flash. By default OpenOCD reads partition table at 0x8000 and uses mappings from the first found application image, but there can be the cases when it will not work, e.g. partition table is not at standard flash location or even there can be multiple images: one factory and two OTA and you may want to debbug any of them. To cover all possible debugging scenarios OpenOCD supports special command which can be used to set arbitrary location of application image to debug. The command has the following format:

```
esp appimage_offset <offset>
```

Offset should be in hex format. To reset to the default behaviour you can specify `-1` as offset.

Note: Since GDB requests memory map from OpenOCD only once when connecting to it, this command should be specified in one of the TCL configuration files, or passed to OpenOCD via its command line. In the latter case command line should look like below:

```
openocd -f board/esp32-wrover-kit-3.3v.cfg -c "init; halt; esp appimage_offset␣
↪0x210000"
```

Another option is to execute that command via OpenOCD telnet session and then connect GDB, but it seems to be less handy.

Why stepping with "next" does not bypass subroutine calls? When stepping through the code with `next` command, GDB is internally setting a breakpoint ahead in the code to bypass the subroutine calls. If all 2 breakpoints are already set, this functionality will not work. If this is the case, delete breakpoints to have one "spare". With all breakpoints already used, stepping through the code with `next` command will work as like with `step` command and debugger will step inside subroutine calls.

Support options for OpenOCD at compile time ESP-IDF has some support options for OpenOCD debugging which can be set at compile time:

- *CONFIG_ESP_DEBUG_OCDAWARE* is enabled by default. If a panic or unhandled exception is thrown and a JTAG debugger is connected (ie OpenOCD is running), ESP-IDF will break into the debugger.
- *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK* (disabled by default) sets watchpoint index 1 (the second of two) at the end of any task stack. This is the most accurate way to debug task stack overflows. Click the link for more details.

Please see the *project configuration menu* menu for more details on setting compile-time options.

FreeRTOS support OpenOCD has explicit support for the ESP-IDF FreeRTOS. GDB can see FreeRTOS tasks as threads. Viewing them all can be done using the GDB `i threads` command, changing to a certain task is done with `thread n`, with `n` being the number of the thread. FreeRTOS detection can be disabled in target's configuration. For more details see *Configuration of OpenOCD for specific target*.

Why to set SPI flash voltage in OpenOCD configuration? The MTDI pin of ESP32, being among four pins used for JTAG communication, is also one of ESP32's bootstrapping pins. On power up ESP32 is sampling binary level on MTDI to set it's internal voltage regulator used to supply power to external SPI flash chip. If binary level on MDTI pin on power up is low, the voltage regulator is set to deliver 3.3 V, if it is high, then the voltage is set to 1.8 V. The MTDI pin should have a pull-up or may rely on internal weak pull down resistor (see ESP32 Series Datasheet for details), depending on the type of SPI chip used. Once JTAG is connected, it overrides the pull-up or pull-down resistor that is supposed to do the bootstrapping.

To handle this issue OpenOCD's board configuration file (e.g. `board\esp32-wrover-kit-3.3v.cfg` for ESP-WROVER-KIT board) provides `ESP32_FLASH_VOLTAGE` parameter to set the idle state of the `TDO` line to a specified binary level, therefore reducing the chance of a bad bootup of application due to incorrect flash voltage.

Check specification of ESP32 module connected to JTAG, what is the power supply voltage of SPI flash chip. Then set `ESP32_FLASH_VOLTAGE` accordingly. Most WROOM modules use 3.3 V flash. WROVER earlier than ESP32-WROVER-B use 1.8 V flash, while ESP32-WROVER-B and -E modules use 3.3 V flash.

Optimize JTAG speed In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG. To do so use the following tips.

1. The upper limit of JTAG clock frequency is 20 MHz if CPU runs at 80 MHz, or 26 MHz if CPU runs at 160 MHz or 240 MHz.
2. Depending on particular JTAG adapter and the length of connecting cables, you may need to reduce JTAG frequency below 20 / 26 MHz.

3. In particular reduce frequency, if you get DSR/DIR errors (and they do not relate to OpenOCD trying to read from a memory range without physical memory being present there).
4. ESP-WROVER-KIT operates stable at 20 / 26 MHz.

What is the meaning of debugger' s startup commands? On startup, debugger is issuing sequence of commands to reset the chip and halt it at specific line of code. This sequence (shown below) is user defined to pick up at most convenient / appropriate line and start debugging.

- `set remote hardware-watchpoint-limit 2` —Restrict GDB to using available hardware watchpoints supported by the chip, 2 for ESP32. For more information see https://sourceware.org/gdb/onlinedocs/gdb/Remote-Configuration.html.
- `mon reset halt` —reset the chip and keep the CPUs halted
- `flushregs` —monitor (`mon`) command can not inform GDB that the target state has changed. GDB will assume that whatever stack the target had before `mon reset halt` will still be valid. In fact, after reset the target state will change, and executing `flushregs` is a way to force GDB to get new state from the target.
- `thb app_main` —insert a temporary hardware breakpoint at `app_main`, put here another function name if required
- `c` —resume the program. It will then stop at breakpoint inserted at `app_main`.

Configuration of OpenOCD for specific target There are several kinds of OpenOCD configuration files (`*.cfg`). All configuration files are located in subdirectories of `share/openocd/scripts` directory of OpenOCD distribution (or `tcl/scripts` directory of the source repository). For the purposes of this guide, the most important ones are `board`, `interface` and `target`.

- `interface` configuration files describe the JTAG adapter. Examples of JTAG adapters are ESP-Prog and J-Link.
- `target` configuration files describe specific chips, or in some cases, modules.
- `board` configuration files are provided for development boards with a built-in JTAG adapter. Such files include an `interface` configuration file to choose the adapter, and `target` configuration file to choose the chip/module.

The following configuration files are available for ESP32:

Table 24: OpenOCD configuration files for ESP32

Name	Description
`board/esp32-wrover-kit-3.3v.cfg`	Board configuration file for ESP-WROVER-KIT with a 3.3 V ESP32-WROOM-32 module or ESP32-WROVER-B / ESP32-WROVER-E module.
`board/esp32-wrover-kit-1.8v.cfg`	Board configuration file for ESP-WROVER-KIT with an 1.8 V ESP32-WROVER module.
`board/esp32-ethernet-kit-3v.cfg`	Board configuration file for ESP-Ethernet-KIT with a 3.3 V ESP32-WROVER-B / ESP32-WROVER-E module.
`target/esp32.cfg`	ESP32 target configuration file. Can be used together with one of the `interface/` configuration files.
`target/esp32-solo-1.cfg`	Target configuration file for ESP32-SOLO-1 module. Different from `esp32.cfg` in that it only configures one CPU.
`interface/ftdi/esp32_devkitj_v1.cfg`	JTAG adapter configuration file for ESP-WROVER-KIT and ESP-Prog boards.

If you are using one of the boards which have a pre-defined configuration file, you only need to pass one `-f` argument to OpenOCD, specifying that file.

If you are using a board not listed here, you need to specify both the interface configuration file and target configuration file.

Custom configuration files OpenOCD configuration files are written in TCL, and include a variety of choices for customization and scripting. This can be useful for non-standard debugging situations. Please refer to OpenOCD Manual for the TCL scripting reference.

OpenOCD configuration variables The following variables can be optionally set before including the ESP-specific target configuration file. This can be done either in a custom configuration file, or from the command line.

The syntax for setting a variable in TCL is:

```
set VARIABLE_NAME value
```

To set a variable from the command line (replace the name of .cfg file with the correct file for your board):

```
openocd -c 'set VARIABLE_NAME value' -f board/esp-xxxxx-kit.cfg
```

It is important to set the variable before including the ESP-specific configuration file, otherwise the variable will not have effect. You can set multiple variables by repeating the `-c` option.

Table 25: Common ESP-related OpenOCD variables

Variable	Description
`ESP_RTOS`	Set to `none` to disable RTOS support. In this case, thread list will not be available in GDB. Can be useful when debugging FreeRTOS itself, and stepping through the scheduler code.
`ESP_FLASH_SIZE`	Set to `0` to disable Flash breakpoints support.
`ESP_SEMIHOST_BASEDIR`	Set to the path (on the host) which will be the default directory for semihosting functions.

Table 26: ESP32-specific OpenOCD variables

Name	Description
`ESP32_FLASH_VOLTAGE`	When using 1.8 V flash ESP32 based modules, set this variable to `1.8`. Refer to *Why to set SPI flash voltage in OpenOCD configuration?*.
`ESP32_ONLYCPU`	For multi-core targets, can be set to `1` to only enable single core debugging.

How debugger resets ESP32? The board can be reset by entering `mon reset` or `mon reset halt` into GDB.

Do not use JTAG pins for something else Operation of JTAG may be disturbed, if some other h/w is connected to JTAG pins besides ESP32 module and JTAG adapter. ESP32 JTAG is using the following pins:

Table 27: ESP32 JTAG pins

ESP32 Pin	JTAG Signal
MTDO / GPIO15	TDO
MTDI / GPIO12	TDI
MTCK / GPIO13	TCK
MTMS / GPIO14	TMS

JTAG communication will likely fail, if configuration of JTAG pins is changed by user application. If OpenOCD initializes correctly (detects the two Tensilica cores), but loses sync and spews out a lot of DTR/DIR errors when the program is ran, it is likely that the application reconfigures the JTAG pins to something else, or the user forgot to connect Vtar to a JTAG adapter that needed it.

Below is an excerpt from series of errors reported by GDB after the application stepped into the code that reconfigured MTDO pin to be an input:

```
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated␣
↪an exception!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated␣
↪an overrun!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated␣
↪an exception!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated␣
↪an overrun!
```

JTAG with Flash Encryption or Secure Boot By default, enabling Flash Encryption and/or Secure Boot will disable JTAG debugging. On first boot, the bootloader will burn an eFuse bit to permanently disable JTAG at the same time it enables the other features.

The project configuration option *CONFIG_SECURE_BOOT_ALLOW_JTAG* will keep JTAG enabled at this time, removing all physical security but allowing debugging. (Although the name suggests Secure Boot, this option can be applied even when only Flash Encryption is enabled).

However, OpenOCD may attempt to automatically read and write the flash in order to set *software breakpoints*. This has two problems:

- Software breakpoints are incompatible with Flash Encryption, OpenOCD currently has no support for encrypting or decrypting flash contents.
- If Secure Boot is enabled, setting a software breakpoint will change the digest of a signed app and make the signature invalid. This means if a software breakpoint is set and then a reset occurs, the signature verification will fail on boot.

To disable software breakpoints while using JTAG, add an extra argument `-c 'set ESP_FLASH_SIZE 0'` to the start of the OpenOCD command line, see *OpenOCD configuration variables*.

Note: For the same reason, the ESP-IDF app may fail bootloader verification of app signatures, when this option is enabled and a software breakpoint is set.

JTAG and ESP32-WROOM-32 AT firmware Compatibility Issue The ESP32-WROOM series of modules come pre-flashed with AT firmware. This firmware configures the pins GPIO12 to GPIO15 as SPI slave interface, which makes using JTAG impossible.

To make JTAG available, build new firmware that is not using pins GPIO12 to GPIO15 dedicated to JTAG communication. After that, flash the firmware onto your module. See also *Do not use JTAG pins for something else*.

Reporting issues with OpenOCD / GDB In case you encounter a problem with OpenOCD or GDB programs itself and do not find a solution searching available resources on the web, open an issue in the OpenOCD issue tracker under https://github.com/espressif/openocd-esp32/issues.

1. In issue report provide details of your configuration:
   a. JTAG adapter type, and the chip/module being debugged.
   b. Release of ESP-IDF used to compile and load application that is being debugged.
   c. Details of OS used for debugging.
   d. Is OS running natively on a PC or on a virtual machine?
2. Create a simple example that is representative to observed issue. Describe steps how to reproduce it. In such an example debugging should not be affected by non-deterministic behaviour introduced by the Wi-Fi stack, so problems will likely be easier to reproduce, if encountered once.
3. Prepare logs from debugging session by adding additional parameters to start up commands.
   OpenOCD:

```
openocd -l openocd_log.txt -d3 -f board/esp32-wrover-kit-3.3v.cfg
```

Logging to a file this way will prevent information displayed on the terminal. This may be a good thing taken amount of information provided, when increased debug level `-d3` is set. If you still like to see the log on the screen, then use another command instead:

```
openocd -d3 -f board/esp32-wrover-kit-3.3v.cfg 2>&1 | tee openocd.log
```

Debugger:

```
xtensa-esp32-elf-gdb -ex "set remotelogfile gdb_log.txt" <all other options>
```

Optionally add command `remotelogfile gdb_log.txt` to the `gdbinit` file.

4. Attach both `openocd_log.txt` and `gdb_log.txt` files to your issue report.

4.16.10 Related Documents

Using Debugger

This section covers configuration and running debugger using several methods:

- from *Eclipse*
- from *Command Line*
- using *idf.py debug targets*

Eclipse

Note: It is recommended to first check if debugger works using *idf.py debug targets* or from *Command Line* and then move to using Eclipse.

Debugging functionality is provided out of box in standard Eclipse installation. Another option is to use pluggins like "GDB Hardware Debugging" plugin. We have found this plugin quite convenient and decided to use throughout this guide.

To begin with, install "GDB Hardware Debugging" plugin by opening Eclipse and going to *Help* > *Install* New Software.

Once installation is complete, configure debugging session following steps below. Please note that some of configuration parameters are generic and some are project specific. This will be shown below by configuring debugging for "blink" example project. If not done already, add this project to Eclipse workspace following guidance in Eclipse Plugin. The source of get-started/blink application is available in examples directory of ESP-IDF repository.

1. In Eclipse go to *Run* > *Debug Configuration*. A new window will open. In the window's left pane double click "GDB Hardware Debugging" (or select "GDB Hardware Debugging" and press the "New" button) to create a new configuration.
2. In a form that will show up on the right, enter the "Name:" of this configuration, e.g. "Blink checking".
3. On the "Main" tab below, under "Project:", press "Browse" button and select the "blink" project.
4. In next line "C/C++ Application:" press "Browse" button and select "blink.elf" file. If "blink.elf" is not there, then likely this project has not been build yet. See Eclipse Plugin how to do it.
5. Finally, under "Build (if required) before launching" click "Disable auto build".
   A sample window with settings entered in points 1 - 5 is shown below.
6. Click "Debugger" tab. In field "GDB Command" enter `xtensa-esp32-elf-gdb` to invoke debugger.
7. Change default configuration of "Remote host" by entering `3333` under the "Port number".
   Configuration entered in points 6 and 7 is shown on the following picture.
8. The last tab to that requires changing of default configuration is "Startup". Under "Initialization Commands" uncheck "Reset and Delay (seconds)" and "Halt"". Then, in entry field below, enter the following lines:

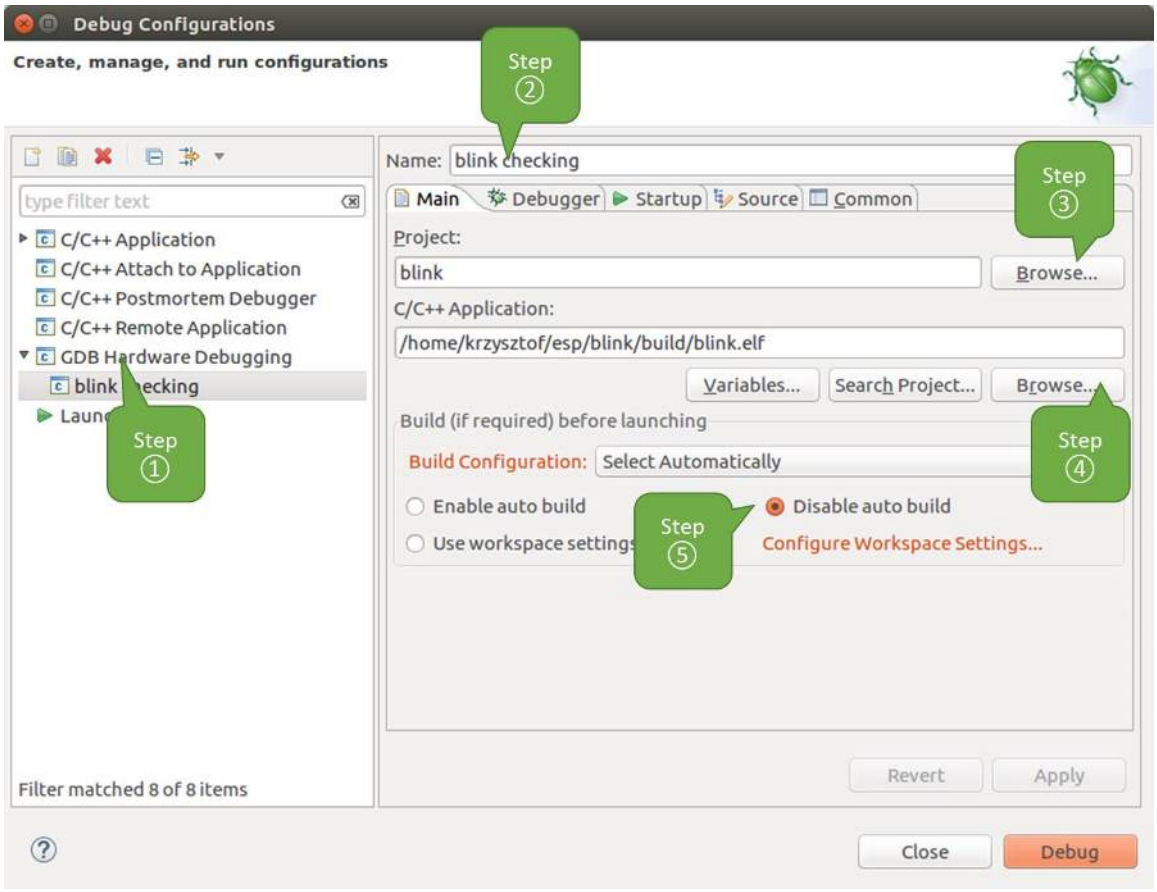


Fig. 38: Configuration of GDB Hardware Debugging - Main tab

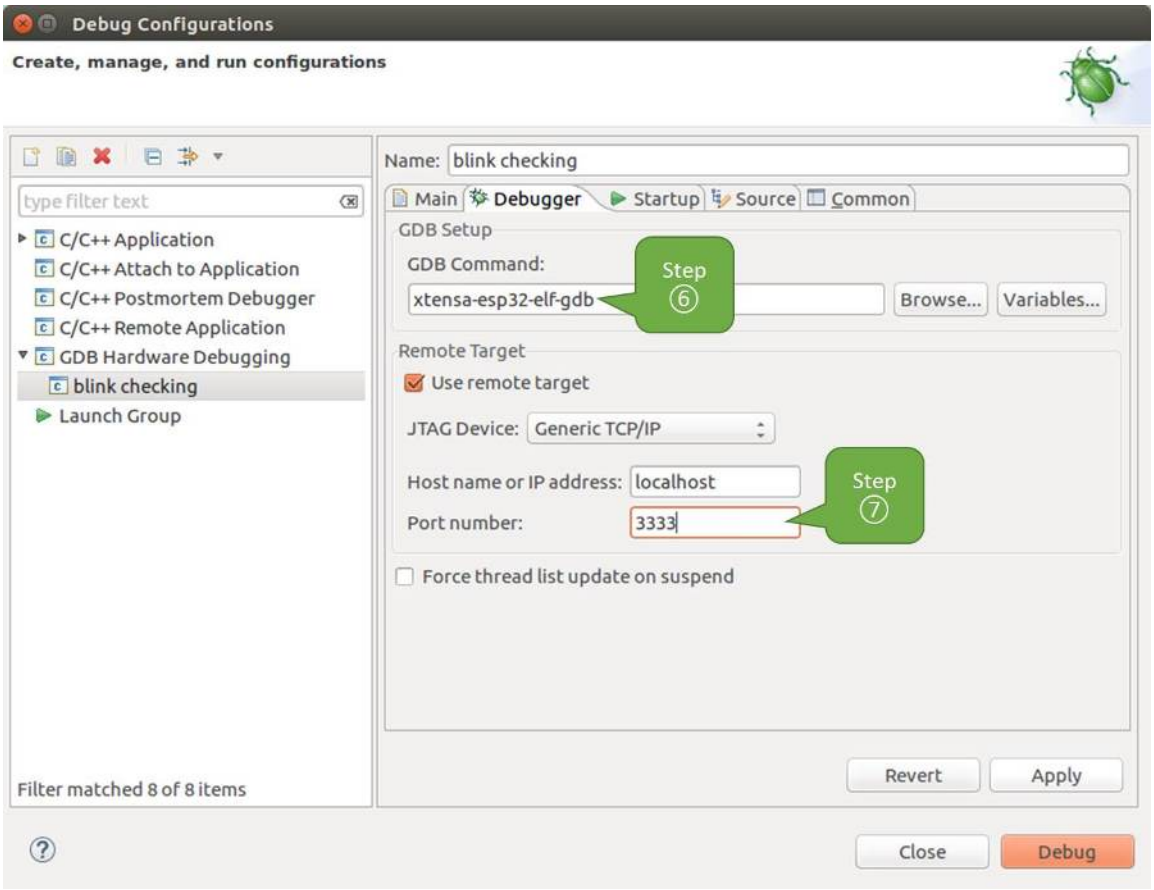


Fig. 39: Configuration of GDB Hardware Debugging - Debugger tab

```
mon reset halt
flushregs
set remote hardware-watchpoint-limit 2
```

Note: If you want to update image in the flash automatically before starting new debug session add the following lines of commands at the beginning of "Initialization Commands" textbox:

```
mon reset halt
mon program_esp ${workspace_loc:blink/build/blink.bin} 0x10000 verify
```

For description of `program_esp` command see *Upload application for debugging*.

9. Under "Load Image and Symbols" uncheck "Load image" option.
10. Further down on the same tab, establish an initial breakpoint to halt CPUs after they are reset by debugger. The plugin will set this breakpoint at the beginning of the function entered under "Set break point at:". Checkout this option and enter `app_main` in provided field.
11. Checkout "Resume" option. This will make the program to resume after `mon reset halt` is invoked per point 8. The program will then stop at breakpoint inserted at `app_main`.
    Configuration described in points 8 - 11 is shown below.

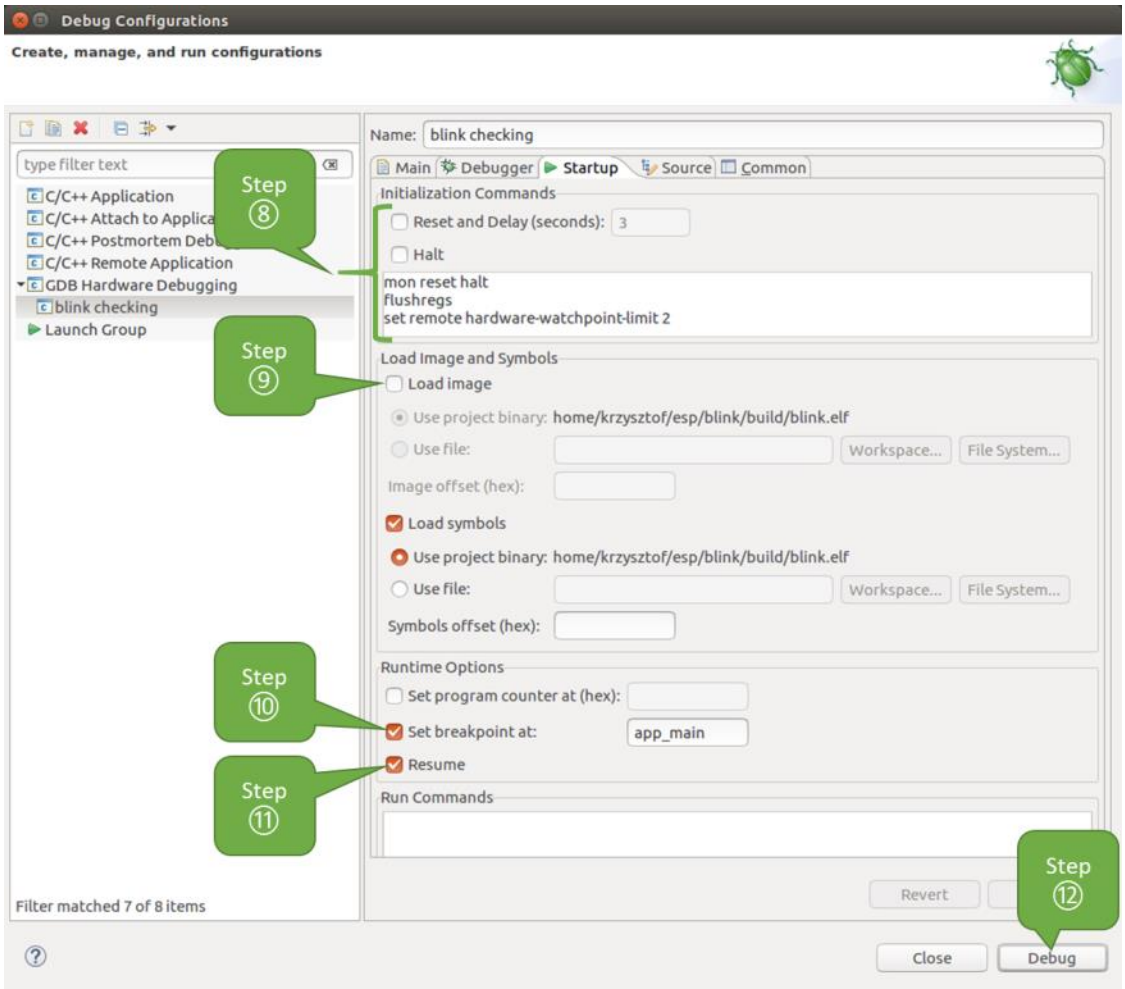


Fig. 40: Configuration of GDB Hardware Debugging - Startup tab

If the "Startup" sequence looks convoluted and respective "Initialization Commands" are not clear to you, check *What is the meaning of debugger's startup commands?* for additional explanation.

12. If you previously completed *Configuring ESP32 Target* steps described above, so the target is running and ready to talk to debugger, go right to debugging by pressing "Debug" button. Otherwise press "Apply" to save changes, go back to *Configuring ESP32 Target* and return here to start debugging.

Once all 1 - 12 configuration steps are satisfied, the new Eclipse perspective called "Debug" will open as shown on example picture below.

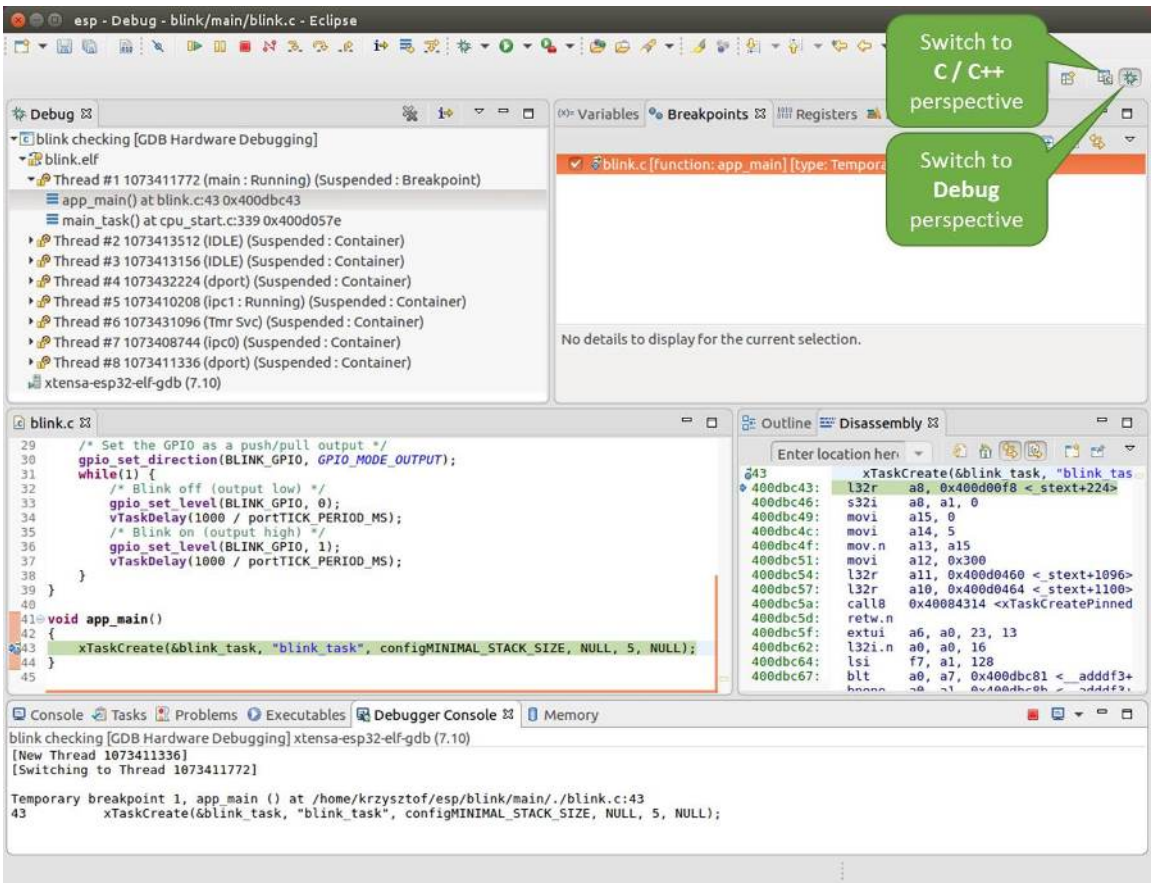


Fig. 41: Debug Perspective in Eclipse

If you are not quite sure how to use GDB, check *Eclipse* example debugging session in section *Debugging Examples*.

Command Line

1. Begin with completing steps described under *Configuring ESP32 Target*. This is prerequisite to start a debugging session.
2. Open a new terminal session and go to directory that contains project for debugging, e.g.

```
cd ~/esp/blink
```

3. When launching a debugger, you will need to provide couple of configuration parameters and commands. Instead of entering them one by one in command line, create a configuration file and name it `gdbinit`:

```
target remote :3333
set remote hardware-watchpoint-limit 2
mon reset halt
flushregs
thb app_main
c
```

   Save this file in current directory.
   For more details what's inside `gdbinit` file, see *What is the meaning of debugger's startup commands?*

4. Now you are ready to launch GDB. Type the following in terminal:

```
xtensa-esp32-elf-gdb -x gdbinit build/blink.elf
```

5. If previous steps have been done correctly, you will see a similar log concluded with `(gdb)` prompt:

```
user-name@computer-name:~/esp/blink$ xtensa-esp32-elf-gdb -x gdbinit build/
↪blink.elf
GNU gdb (crosstool-NG crosstool-ng-1.22.0-61-gab8375a) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_pc-linux-gnu --target=xtensa-
↪esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/blink.elf...done.
0x400d10d8 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/
↪components/esp32/./freertos_hooks.c:52
52          asm("waiti 0");
JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),␣
↪part: 0x2003, ver: 0x1)
JTAG tap: esp32.slave tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),␣
↪part: 0x2003, ver: 0x1)
esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x5000004B (active)    APP_CPU: PC=0x00000000
esp32: target state: halted
esp32: Core was reset (pwrstat=0x1F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x40000400 (active)    APP_CPU: PC=0x40000400
esp32: target state: halted
Hardware assisted breakpoint 1 at 0x400db717: file /home/user-name/esp/blink/
↪main/./blink.c, line 43.
0x0:    0x00000000
Target halted. PRO_CPU: PC=0x400DB717 (active)    APP_CPU: PC=0x400D10D8
[New Thread 1073428656]
[New Thread 1073413708]
[New Thread 1073431316]
[New Thread 1073410672]
[New Thread 1073408876]
[New Thread 1073432196]
[New Thread 1073411552]
[Switching to Thread 1073411996]

Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main/./blink.
↪c:43
43      xTaskCreate(&blink_task, "blink_task", 512, NULL, 5, NULL);
(gdb)
```

Note the third line from bottom that shows debugger halting at breakpoint established in `gdbinit` file at function `app_main()`. Since the processor is halted, the LED should not be blinking. If this is what you see as well, you are ready to start debugging.

If you are not quite sure how to use GDB, check *Command Line* example debugging session in section *Debugging Examples*.

idf.py debug targets It is also possible to execute the described debugging tools conveniently from `idf.py`. These commands are supported:

1. `idf.py openocd`
   Runs OpenOCD in a console with configuration defined in the environment or via command line. It uses default script directory defined as `OPENOCD_SCRIPTS` environmental variable, which is automatically added from an Export script (`export.sh` or `export.bat`). It is possible to override the script location using command line argument `--openocd-scripts`.
   As for the JTAG configuration of the current board, please use the environmental variable `OPENOCD_COMMANDS` or `--openocd-commands` command line argument. If none of the above is defined, OpenOCD is started with `-f board/esp32-wrover-kit-3.3v.cfg` board definition.
2. `idf.py gdb`
   Starts the gdb the same way as the *Command Line*, but generates the initial gdb scripts referring to the current project elf file.
3. `idf.py gdbtui`
   The same as *2*, but starts the gdb with `tui` argument allowing very simple source code view.
4. `idf.py gdbgui`
   Starts gdbgui debugger frontend enabling out-of-the-box debugging in a browser window. Please run the install script with the "–enable-gdbgui" argument in order to make this option supported, e.g. `install.sh --enable-gdbgui`.
   It is possible to combine these debugging actions on a single command line allowing convenient setup of blocking and non-blocking actions in one step. `idf.py` implements a simple logic to move the background actions (such as openocd) to the beginning and the interactive ones (such as gdb, monitor) to the end of the action list. An example of a very useful combination is:

   ```
   idf.py openocd gdbgui monitor
   ```

   The above command runs OpenOCD in the background, starts gdbgui to open a browser window with active debugger frontend and opens a serial monitor in the active console.

Debugging Examples

This section describes debugging with GDB from *Eclipse* as well as from *Command Line*.

Eclipse Verify if your target is ready and loaded with get-started/blink example. Configure and start debugger following steps in section *Eclipse*. Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`.

Examples in this section

1. *Navigating through the code, call stack and threads*
2. *Setting and clearing breakpoints*
3. *Halting the target manually*
4. *Stepping through the code*
5. *Checking and setting memory*
6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Navigating through the code, call stack and threads When the target is halted, debugger shows the list of threads in "Debug" window. The line of code where program halted is highlighted in another window below, as shown on the following picture. The LED stops blinking.

Specific thread where the program halted is expanded showing the call stack. It represents function calls that lead up to the highlighted line of code, where the target halted. The first line of call stack under Thread #1 contains the last called function `app_main()`, that in turn was called from function `main_task()` shown in a line below. Each

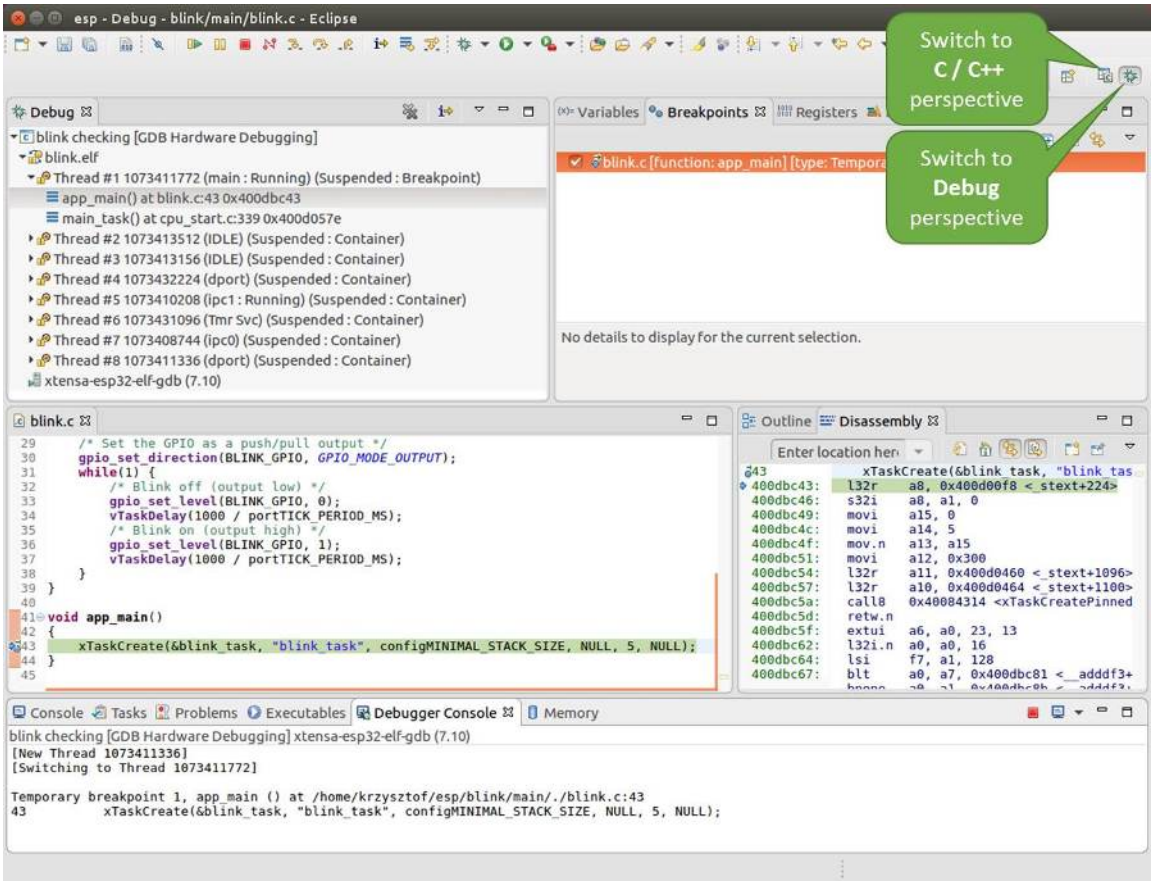


Fig. 42: Debug Perspective in Eclipse

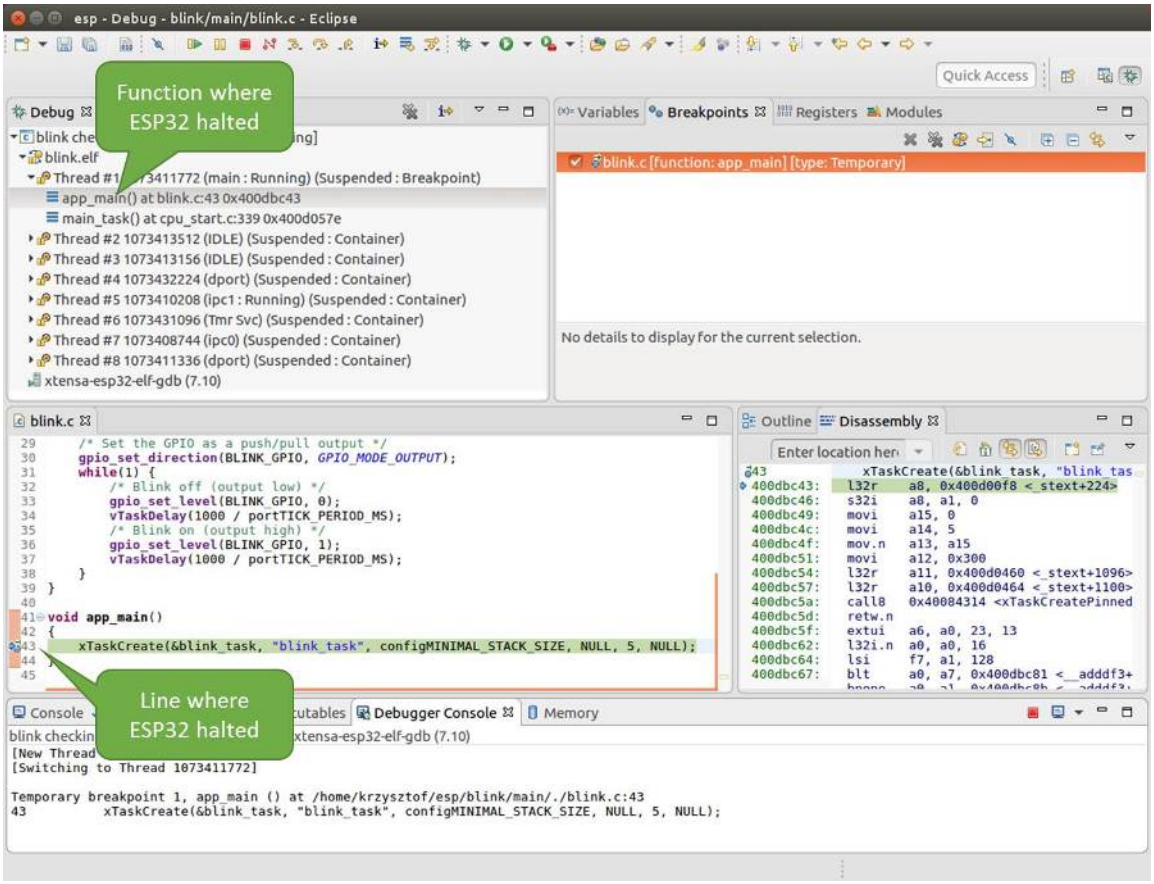


Fig. 43: Target halted during debugging

line of the stack also contains the file name and line number where the function was called. By clicking / highlighting the stack entries, in window below, you will see contents of this file.

By expanding threads you can navigate throughout the application. Expand Thread #5 that contains much longer call stack. You will see there, besides function calls, numbers like `0x4000000c`. They represent addresses of binary code not provided in source form.

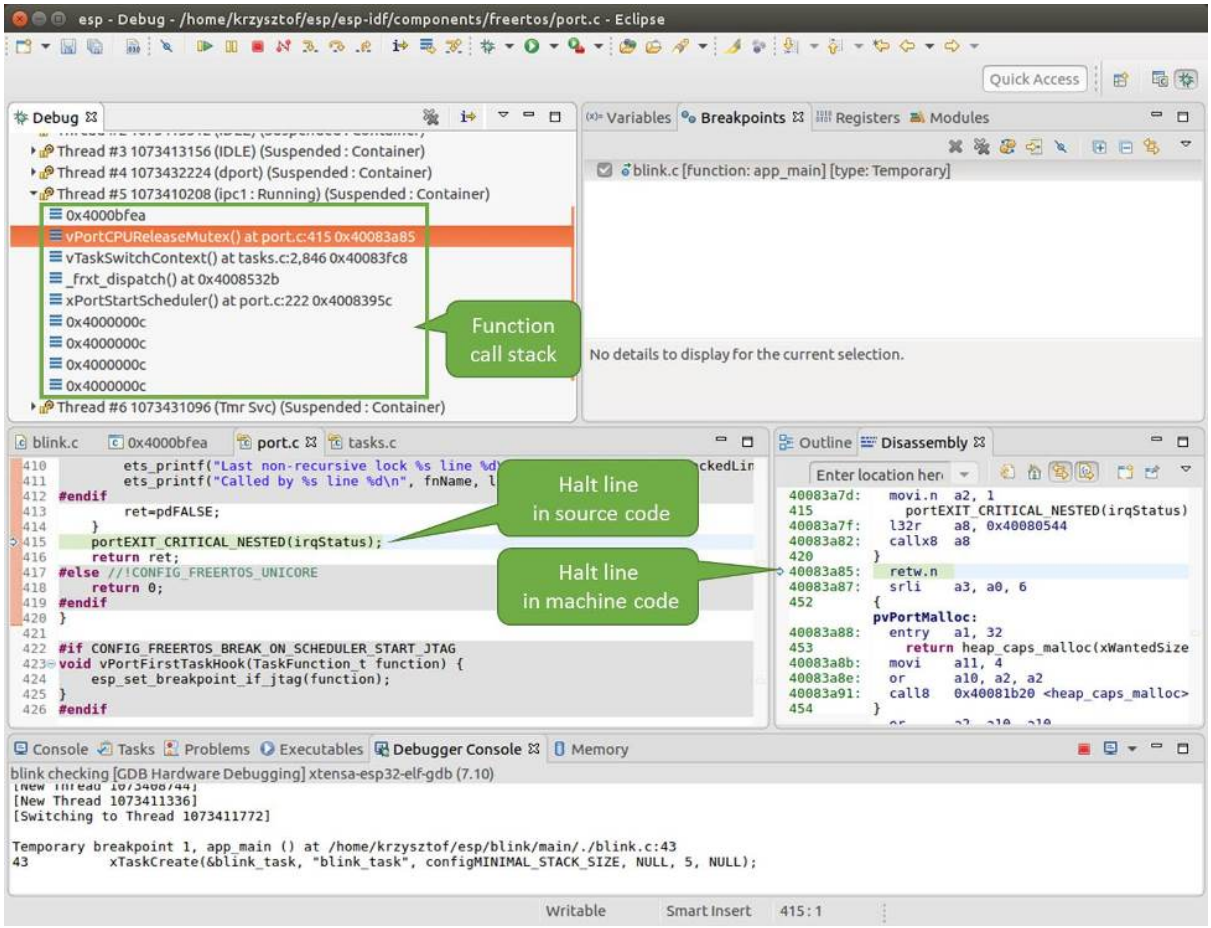


Fig. 44: Navigate through the call stack

In another window on right, you can see the disassembled machine code no matter if your project provides it in source or only the binary form.

Go back to the `app_main()` in Thread #1 to familiar code of `blink.c` file that will be examined in more details in the following examples. Debugger makes it easy to navigate through the code of entire application. This comes handy when stepping through the code and working with breakpoints and will be discussed below.

Setting and clearing breakpoints When debugging, we would like to be able to stop the application at critical lines of code and then examine the state of specific variables, memory and registers / peripherals. To do so we are using breakpoints. They provide a convenient way to quickly get to and halt the application at specific line.

Let's establish two breakpoints when the state of LED changes. Basing on code listing above, this happens at lines 33 and 36. To do so, hold the "Control" on the keyboard and double clink on number `33` in file `blink.c` file. A dialog will open where you can confirm your selection by pressing "OK" button. If you do not like to see the dialog just double click the line number. Set another breakpoint in line 36.

Information how many breakpoints are set and where is shown in window "Breakpoints" on top right. Click "Show Breakpoints Supported by Selected Target" to refresh this list. Besides the two just set breakpoints the list may contain temporary breakpoint at function `app_main()` established at debugger start. As maximum two breakpoints are allowed (see *Breakpoints and watchpoints available*), you need to delete it, or debugging will fail.

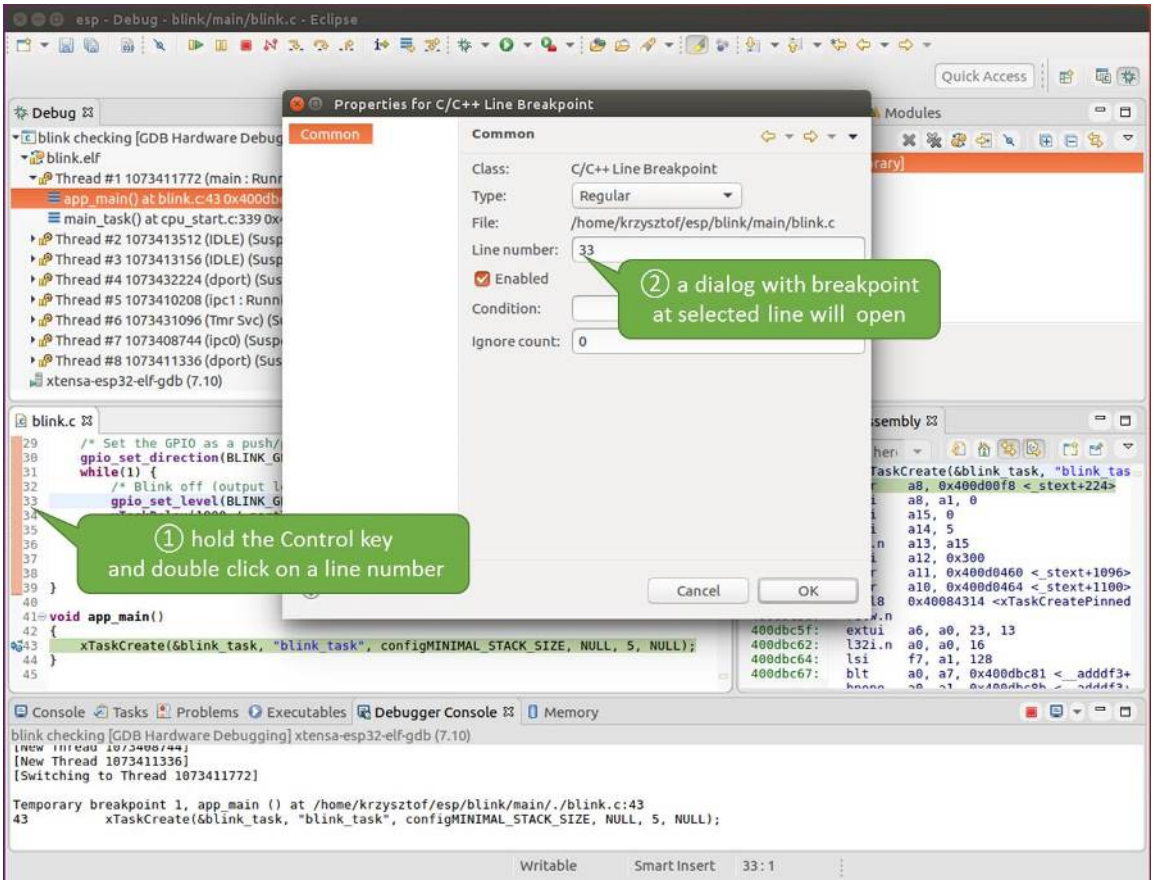


Fig. 45: Setting a breakpoint

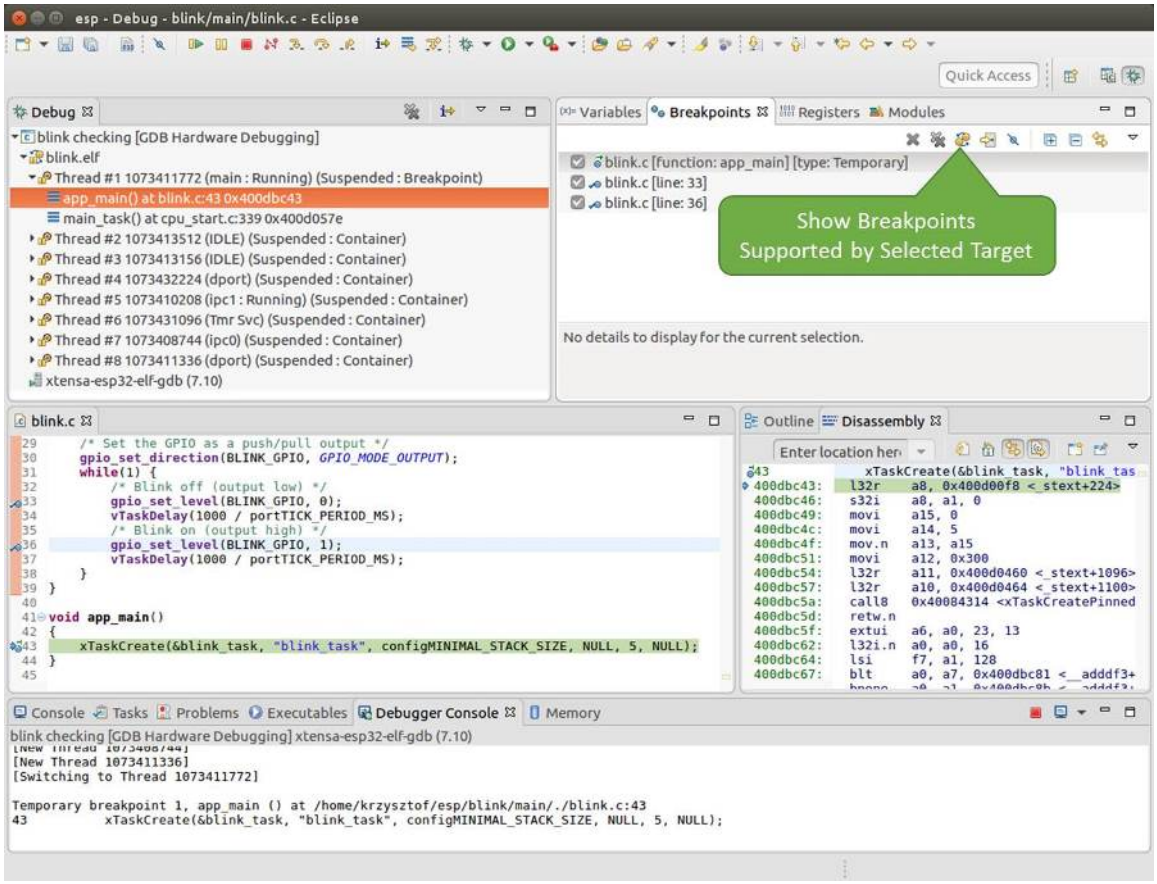


Fig. 46: Three breakpoints are set / maximum two are allowed

If you now click "Resume" (click `blink_task()` under "Tread #8", if "Resume" button is grayed out), the processor will run and halt at a breakpoint. Clicking "Resume" another time will make it run again, halt on second breakpoint, and so on.

You will be also able to see that LED is changing the state after each click to "Resume" program execution.

Read more about breakpoints under *Breakpoints and watchpoints available* and *What else should I know about breakpoints?*

Halting the target manually When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by pressing "Suspend" button.

To check it, delete all breakpoints and click "Resume". Then click "Suspend". Application will be halted at some random point and LED will stop blinking. Debugger will expand tread and highlight the line of code where application halted.

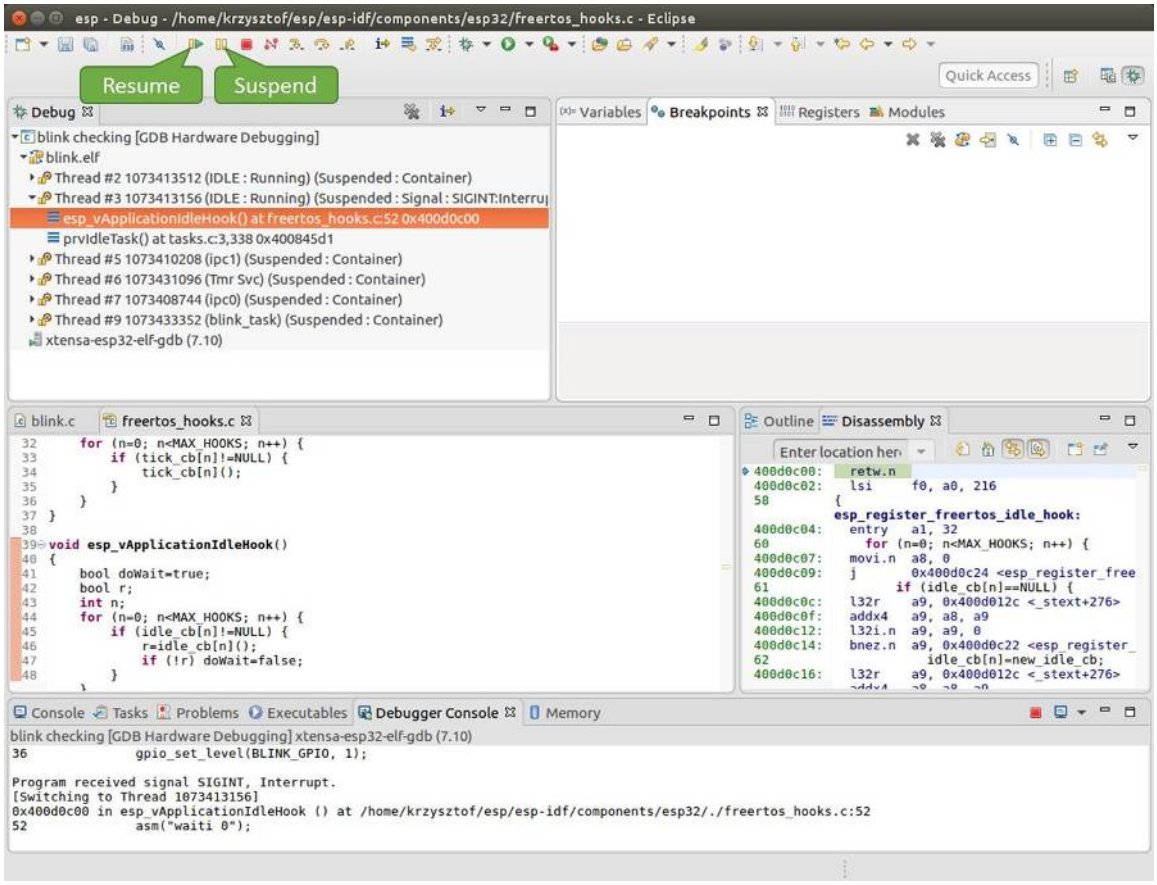


Fig. 47: Target halted manually

In particular case above, the application has been halted in line 52 of code in file `freertos_hooks.c` Now you can resume it again by pressing "Resume" button or do some debugging as discussed below.

Stepping through the code It is also possible to step through the code using "Step Into (F5)" and "Step Over (F6)" commands. The difference is that "Step Into (F5)" is entering inside subroutines calls, while "Step Over (F6)" steps over the call, treating it as a single source line.

Before being able to demonstrate this functionality, using information discussed in previous paragraph, make sure that you have only one breakpoint defined at line `36` of `blink.c`.

Resume program by entering pressing F8 and let it halt. Now press "Step Over (F6)", one by one couple of times, to see how debugger is stepping one program line at a time.

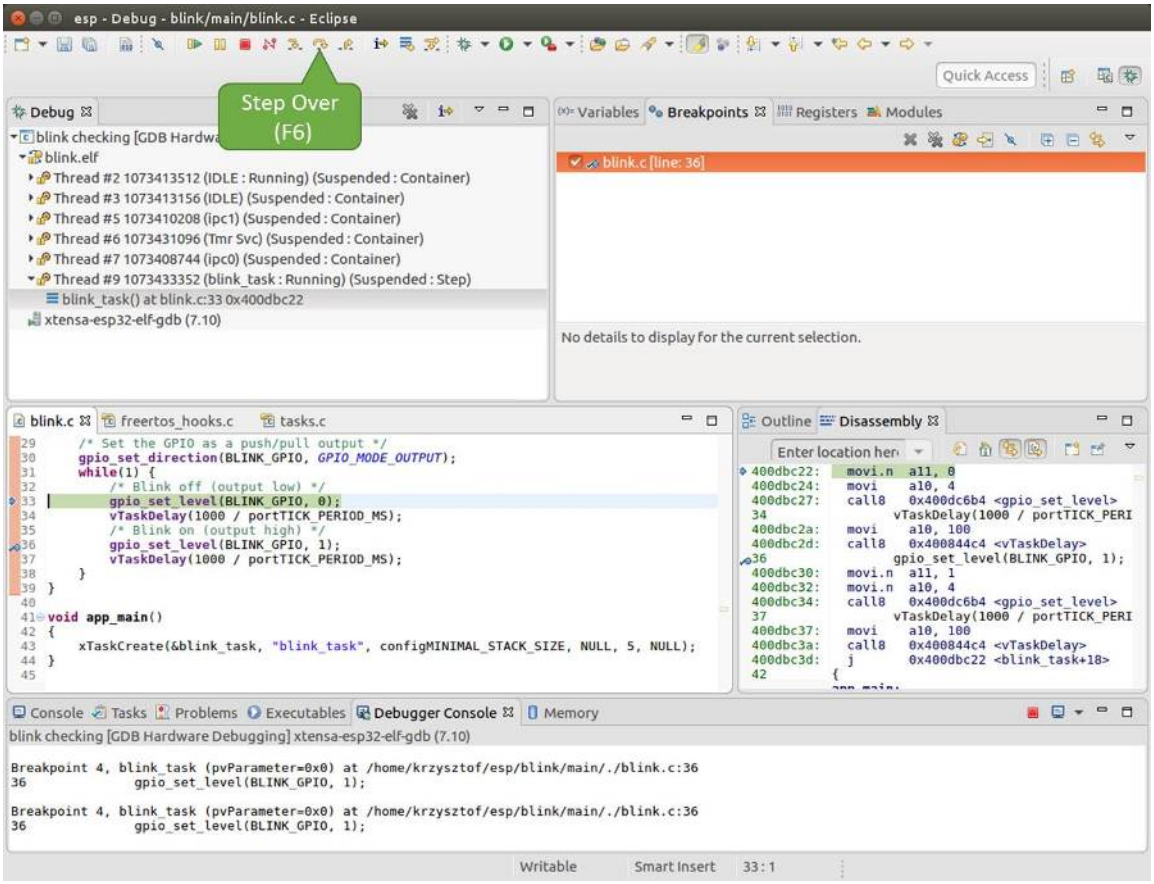


Fig. 48: Stepping through the code with "Step Over (F6)"

If you press “Step Into (F5)” instead, then debugger will step inside subroutine calls.

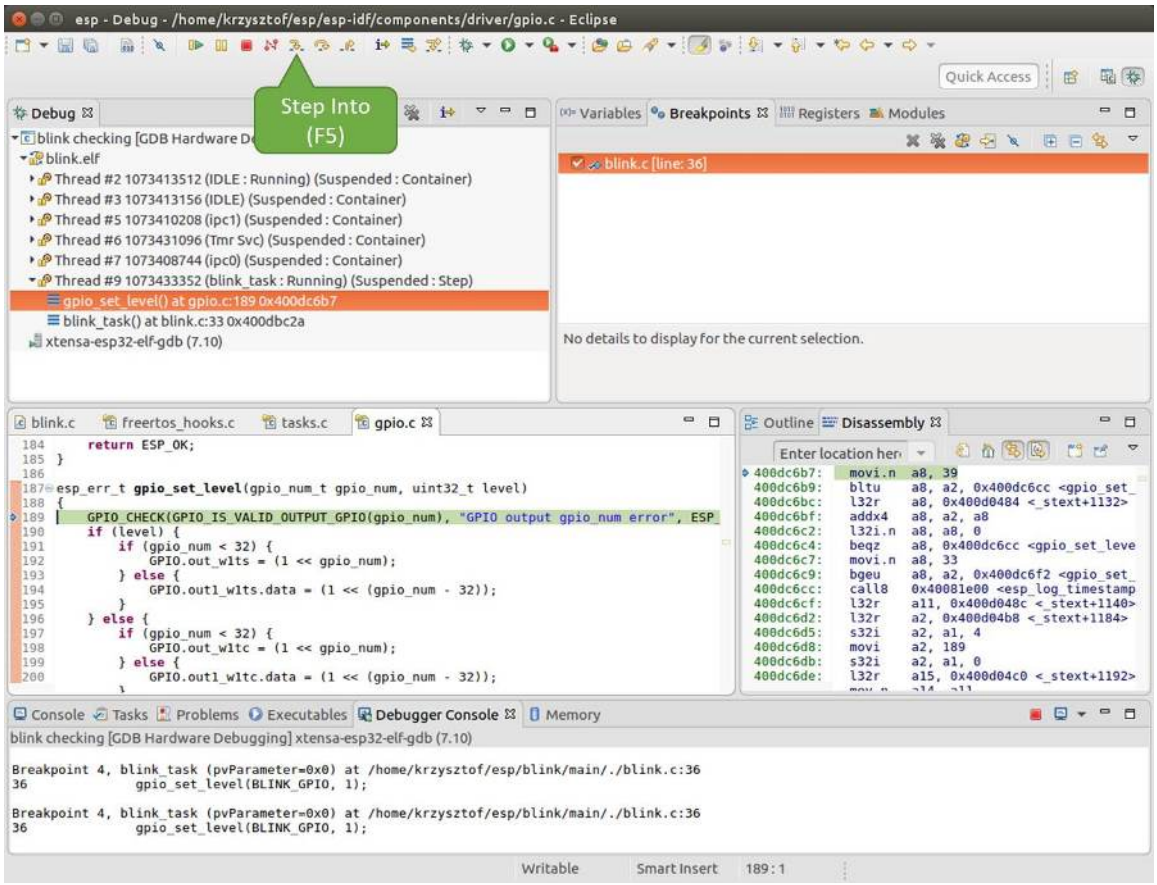


Fig. 49: Stepping through the code with “Step Into (F5)”

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See *Why stepping with “next” does not bypass subroutine calls?* for potential limitation of using `next` command.

Checking and setting memory To display or set contents of memory use “Memory” tab at the bottom of “Debug” perspective.

With the “Memory” tab, we will read from and write to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO’s.

For more information, see *ESP32 Technical Reference Manual > IO MUX and GPIO Matrix (GPIO, IO_MUX)* [PDF].

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Click “Memory” tab and then “Add Memory Monitor” button. Enter `0x3FF44004` in provided dialog.

Now resume program by pressing F8 and observe “Monitor” tab.

You should see one bit being flipped over at memory location `0x3FF44004` (and LED changing the state) each time F8 is pressed.

To set memory use the same “Monitor” tab and the same memory location. Type in alternate bit pattern as previously observed. Immediately after pressing enter you will see LED changing the state.

Watching and setting program variables A common debugging tasks is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `while(1)` of this function to get `i` incremented on each blink.

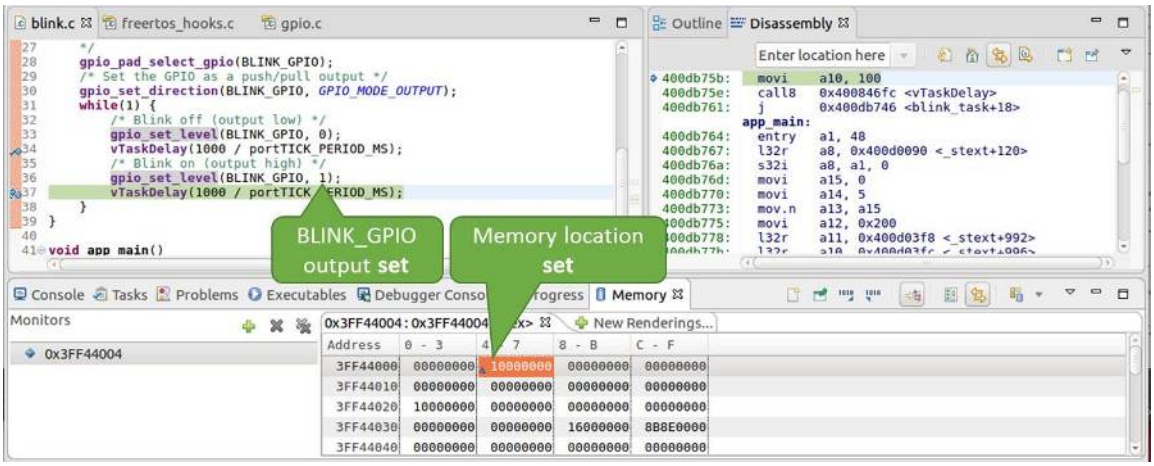


Fig. 50: Observing memory location 0x3FF44004 changing one bit to "ON"

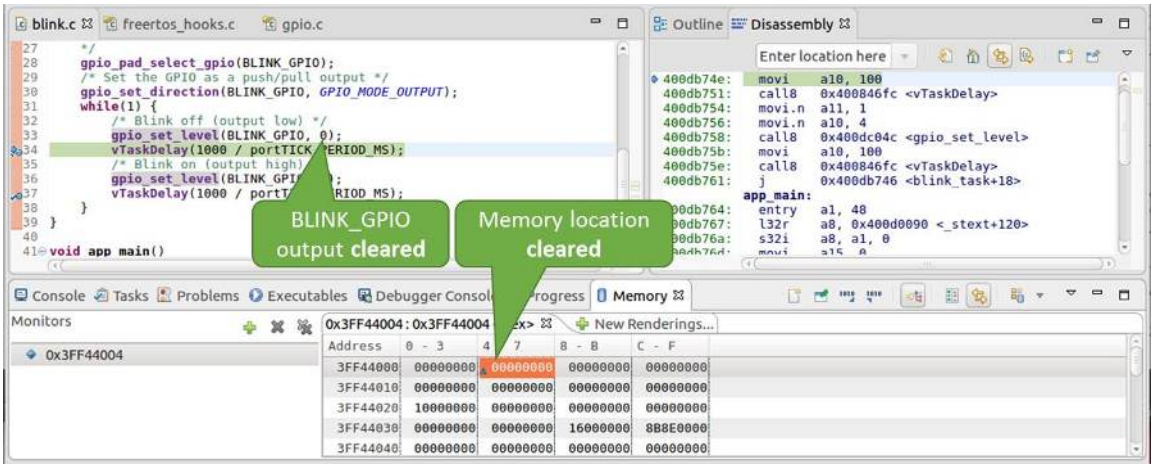


Fig. 51: Observing memory location 0x3FF44004 changing one bit to "OFF"

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter a breakpoint in the line where you put `i++`.

In next step, in the window with "Breakpoints", click the "Expressions" tab. If this tab is not visible, then add it by going to the top menu Window > Show View > Expressions. Then click "Add new expression" and enter `i`.

Resume program execution by pressing F8. Each time the program is halted you will see `i` value being incremented.

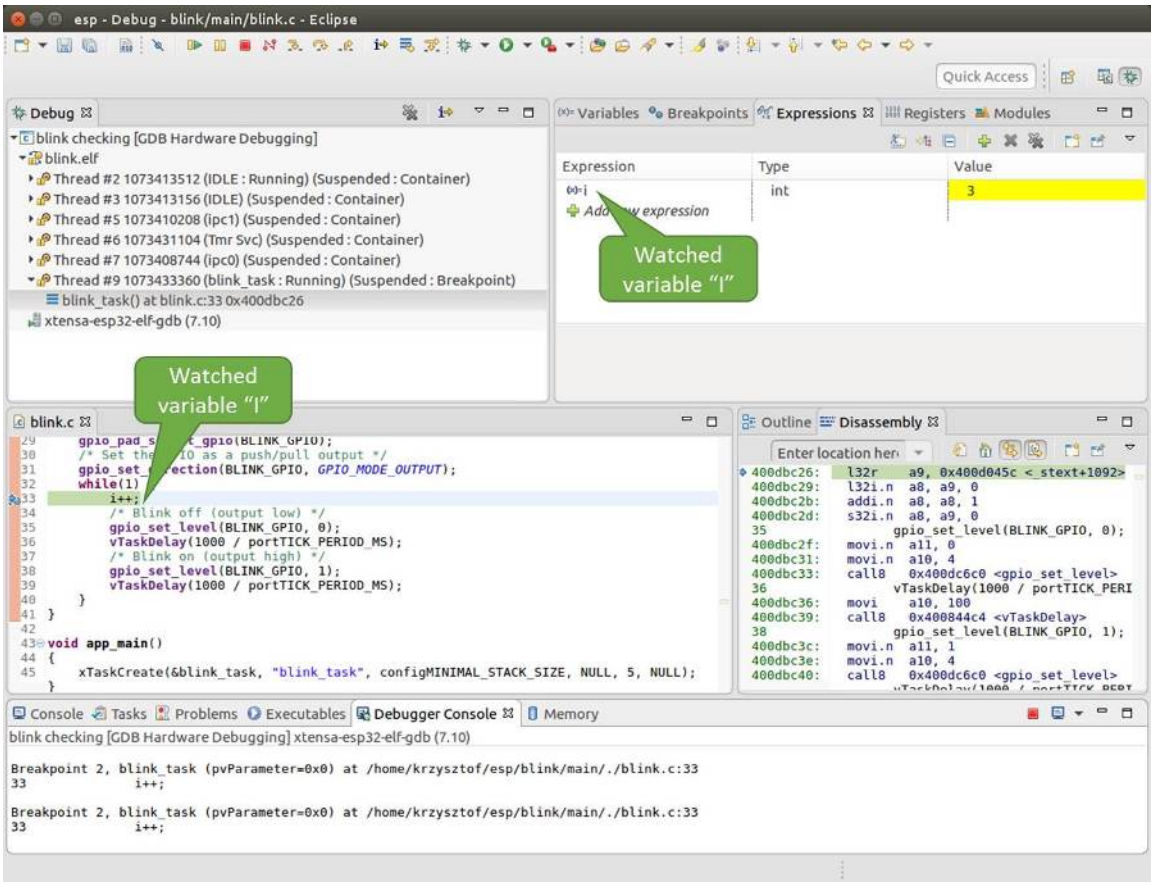


Fig. 52: Watching program variable "i"

To modify `i` enter a new number in "Value" column. After pressing "Resume (F8)" the program will keep incrementing `i` starting from the new entered number.

Setting conditional breakpoints Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Right click on the breakpoint to open a context menu and select "Breakpoint Properties". Change the selection under "Type:" to "Hardware" and enter a "Condition:" like `i == 2`.

If current value of `i` is less than `2` (change it if required) and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt.

Command Line Verify if your target is ready and loaded with get-started/blink example. Configure and start debugger following steps in section *Command Line*. Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`:

```
Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main/./blink.c:43
43      xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5,␣
↪NULL);
(gdb)
```

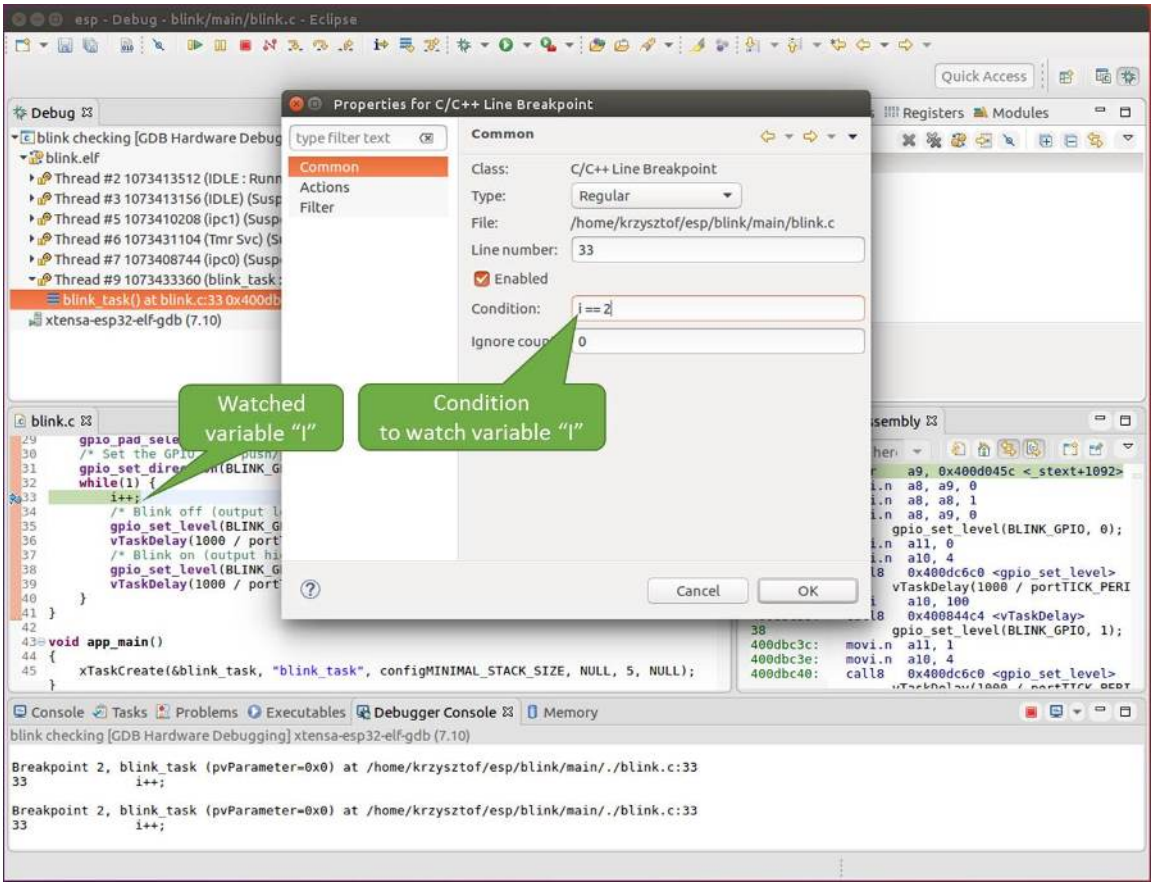


Fig. 53: Setting a conditional breakpoint

Examples in this section

1. *Navigating through the code, call stack and threads*
2. *Setting and clearing breakpoints*
3. *Halting and resuming the application*
4. *Stepping through the code*
5. *Checking and setting memory*
6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Navigating through the code, call stack and threads When you see the `(gdb)` prompt, the application is halted. LED should not be blinking.

To find out where exactly the code is halted, enter `l` or `list`, and debugger will show couple of lines of code around the halt point (line 43 of code in file `blink.c`)

```
(gdb) l
38      }
39  }
40
41  void app_main()
42  {
43      xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5,␣
↪NULL);
44  }
(gdb)
```

Check how code listing works by entering, e.g. `l 30, 40` to see particular range of lines of code.

You can use `bt` or `backtrace` to see what function calls lead up to this code:

```
(gdb) bt
#0  app_main () at /home/user-name/esp/blink/main/./blink.c:43
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↪esp32/./cpu_start.c:339
(gdb)
```

Line #0 of output provides the last function call before the application halted, i.e. `app_main ()` we have listed previously. The `app_main ()` was in turn called by function `main_task` from line 339 of code located in file `cpu_start.c`.

To get to the context of `main_task` in file `cpu_start.c`, enter `frame N`, where N = 1, because the `main_task` is listed under #1):

```
(gdb) frame 1
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↪esp32/./cpu_start.c:339
339     app_main();
(gdb)
```

Enter `l` and this will reveal the piece of code that called `app_main()` (in line 339):

```
(gdb) l
334         ;
335     }
336 #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
```

(continues on next page)

(continued from previous page)

```
342
(gdb)
```

By listing some lines before, you will see the function name `main_task` we have been looking for:

```
(gdb) l 326, 341
326 static void main_task(void* args)
327 {
328     // Now that the application is about to start, disable boot watchdogs
329     REG_CLR_BIT(TIMG_WDTCONFIG0_REG(0), TIMG_WDT_FLASHBOOT_MOD_EN_S);
330     REG_CLR_BIT(RTC_CNTL_WDTCONFIG0_REG, RTC_CNTL_WDT_FLASHBOOT_MOD_EN);
331 #if !CONFIG_FREERTOS_UNICORE
332     // Wait for FreeRTOS initialization to finish on APP CPU, before replacing␣
↪its startup stack
333     while (port_xSchedulerRunning[1] == 0) {
334         ;
335     }
336 #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
(gdb)
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
(gdb) i threads
  Id   Target Id         Frame
  8    Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=
↪<optimized out>)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:170
  7    Thread 1073408744 (ipc0) xQueueGenericReceive (xQueue=0x3ffae694,␣
↪pvBuffer=0x0, xTicksToWait=1644638200,
    xJustPeeking=0) at /home/user-name/esp/esp-idf/components/freertos/./queue.
↪c:1452
  6    Thread 1073431096 (Tmr Svc) prvTimerTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./timers.c:445
  5    Thread 1073410208 (ipc1 : Running) 0x4000bfea in ?? ()
  4    Thread 1073432224 (dport) dport_access_init_core (arg=0x0)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:150
  3    Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
  2    Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
* 1    Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/
↪main/./blink.c:43
(gdb)
```

The thread list shows the last function calls per each thread together with the name of C source file if available.

You can navigate to specific thread by entering `thread N`, where `N` is the thread Id. To see how it works go to thread thread 5:

```
(gdb) thread 5
[Switching to thread 5 (Thread 1073410208)]
#0  0x4000bfea in ?? ()
(gdb)
```

Then check the backtrace:

```
(gdb) bt
#0  0x4000bfea in ?? ()
#1  0x40083a85 in vPortCPUReleaseMutex (mux=<optimized out>) at /home/user-name/
↪esp/esp-idf/components/freertos/./port.c:415
#2  0x40083fc8 in vTaskSwitchContext () at /home/user-name/esp/esp-idf/components/
↪freertos/./tasks.c:2846
#3  0x4008532b in _frxt_dispatch ()
#4  0x4008395c in xPortStartScheduler () at /home/user-name/esp/esp-idf/components/
↪freertos/./port.c:222
#5  0x4000000c in ?? ()
#6  0x4000000c in ?? ()
#7  0x4000000c in ?? ()
#8  0x4000000c in ?? ()
(gdb)
```

As you see, the backtrace may contain several entries. This will let you check what exact sequence of function calls lead to the code where the target halted. Question marks `??` instead of a function name indicate that application is available only in binary format, without any source file in C language. The value like `0x4000bfea` is the memory address of the function call.

Using `bt`, `i threads`, `thread N` and `list` commands we are now able to navigate through the code of entire application. This comes handy when stepping through the code and working with breakpoints and will be discussed below.

Setting and clearing breakpoints When debugging, we would like to be able to stop the application at critical lines of code and then examine the state of specific variables, memory and registers / peripherals. To do so we are using breakpoints. They provide a convenient way to quickly get to and halt the application at specific line.

Let's establish two breakpoints when the state of LED changes. Basing on code listing above this happens at lines 33 and 36. Breakpoints may be established using command `break M` where M is the code line number:

```
(gdb) break 33
Breakpoint 2 at 0x400db6f6: file /home/user-name/esp/blink/main/./blink.c, line 33.
(gdb) break 36
Breakpoint 3 at 0x400db704: file /home/user-name/esp/blink/main/./blink.c, line 36.
```

If you new enter `c`, the processor will run and halt at a breakpoint. Entering `c` another time will make it run again, halt on second breakpoint, and so on:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F6 (active)    APP_CPU: PC=0x400D10D8

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:33
33          gpio_set_level(BLINK_GPIO, 0);
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F8 (active)    APP_CPU: PC=0x400D10D8
Target halted. PRO_CPU: PC=0x400DB704 (active)    APP_CPU: PC=0x400D10D8

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:36
36          gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

You will be also able to see that LED is changing the state only if you resume program execution by entering `c`.

To examine how many breakpoints are set and where, use command `info break`:

```
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x400db6f6 in blink_task at /home/user-name/esp/
↪blink/main/./blink.c:33
    breakpoint already hit 1 time
3       breakpoint     keep y   0x400db704 in blink_task at /home/user-name/esp/
↪blink/main/./blink.c:36
    breakpoint already hit 1 time
(gdb)
```

Please note that breakpoint numbers (listed under `Num`) start with `2`. This is because first breakpoint has been already established at function `app_main()` by running command `thb app_main` on debugger launch. As it was a temporary breakpoint, it has been automatically deleted and now is not listed anymore.

To remove breakpoints enter `delete N` command (in short `d N`), where `N` is the breakpoint number:

```
(gdb) delete 1
No breakpoint number 1.
(gdb) delete 2
(gdb)
```

Read more about breakpoints under *Breakpoints and watchpoints available* and *What else should I know about breakpoints?*

Halting and resuming the application When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by entering Ctrl+C.

To check it delete all breakpoints and enter `c` to resume application. Then enter Ctrl+C. Application will be halted at some random point and LED will stop blinking. Debugger will print the following:

```
(gdb) c
Continuing.
^CTarget halted. PRO_CPU: PC=0x400D0C00             APP_CPU: PC=0x400D0C00 (active)
[New Thread 1073433352]

Program received signal SIGINT, Interrupt.
[Switching to Thread 1073413512]
0x400d0c00 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/
↪components/esp32/./freertos_hooks.c:52
52          asm("waiti 0");
(gdb)
```

In particular case above, the application has been halted in line 52 of code in file `freertos_hooks.c`. Now you can resume it again by enter `c` or do some debugging as discussed below.

Note: In MSYS2 shell Ctrl+C does not halt the target but exists debugger. To resolve this issue consider debugging with *Eclipse* or check a workaround under http://www.mingw.org/wiki/Workaround_for_GDB_Ctrl_C_Interrupt.

Stepping through the code It is also possible to step through the code using `step` and `next` commands (in short `s` and `n`). The difference is that `step` is entering inside subroutines calls, while `next` steps over the call, treating it as a single source line.

To demonstrate this functionality, using command `break` and `delete` discussed in previous paragraph, make sure that you have only one breakpoint defined at line `36` of `blink.c`:

```
(gdb) info break
Num     Type           Disp Enb Address    What
```

(continues on next page)

(continued from previous page)

```
3       breakpoint     keep y   0x400db704 in blink_task at /home/user-name/esp/
↪blink/main/./blink.c:36
    breakpoint already hit 1 time
(gdb)
```

Resume program by entering `c` and let it halt:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB754 (active)    APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:36
36          gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

Then enter `n` couple of times to see how debugger is stepping one program line at a time:

```
(gdb) n
Target halted. PRO_CPU: PC=0x400DB756 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB758 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)    APP_CPU: PC=0x400D1128
37          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) n
Target halted. PRO_CPU: PC=0x400DB75E (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400846FC (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB761 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB746 (active)    APP_CPU: PC=0x400D1128
33          gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

If you enter `s` instead, then debugger will step inside subroutine calls:

```
(gdb) s
Target halted. PRO_CPU: PC=0x400DB748 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74B (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04F (active)    APP_CPU: PC=0x400D1128
gpio_set_level (gpio_num=GPIO_NUM_4, level=0) at /home/user-name/esp/esp-idf/
↪components/driver/./gpio.c:183
183     GPIO_CHECK(GPIO_IS_VALID_OUTPUT_GPIO(gpio_num), "GPIO output gpio_num error
↪", ESP_ERR_INVALID_ARG);
(gdb)
```

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See *Why stepping with "next" does not bypass subroutine calls?* for potential limitation of using `next` command.

Checking and setting memory Displaying the contents of memory is done with command `x`. With additional parameters you may vary the format and count of memory locations displayed. Run `help x` to see more details. Companion command to `x` is `set` that let you write values to the memory.

We will demonstrate how `x` and `set` work by reading from and writing to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO's.

For more information, see *ESP32 Technical Reference Manual* > *IO MUX and GPIO Matrix (GPIO, IO_MUX)* [PDF].

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Enter two times `c` to get to the break point followed by `x /1wx 0x3FF44004` to display contents of `GPIO_OUT_REG` memory location:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB75E (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74E (active)    APP_CPU: PC=0x400D1128

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:34
34          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)    APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:37
37          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000010
(gdb)
```

If your are blinking LED connected to GPIO4, then you should see fourth bit being flipped each time the LED changes the state:

```
0x3ff44004: 0x00000000
...
0x3ff44004: 0x00000010
```

Now, when the LED is off, that corresponds to `0x3ff44004: 0x00000000` being displayed, try using `set` command to set this bit by writting `0x00000010` to the same memory location:

```
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) set {unsigned int}0x3FF44004=0x000010
```

You should see the LED to turn on immediately after entering `set {unsigned int}0x3FF44004=0x000010` command.

Watching and setting program variables A common debugging tasks is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `while(1)` of this function to get `i` incremented on each blink.

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter the command `watch i`:

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb)
```

This will insert so called "watchpoint" in each place of code where variable `i` is being modified. Now enter `continue` to resume the application and observe it being halted:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)    APP_CPU: PC=0x400D0811
[New Thread 1073432196]
```

(continues on next page)

(continued from previous page)

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 1073432196]
0x400db751 in blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:33
33          i++;
(gdb)
```

Resume application couple more times so `i` gets incremented. Now you can enter `print i` (in short `p i`) to check the current value of `i`:

```
(gdb) p i
$1 = 3
(gdb)
```

To modify the value of `i` use `set` command as below (you can then print it out to check if it has been indeed changed):

```
(gdb) set var i = 0
(gdb) p i
$3 = 0
(gdb)
```

You may have up to two watchpoints, see *Breakpoints and watchpoints available*.

Setting conditional breakpoints Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Delete existing breakpoints and try this:

```
(gdb) break blink.c:34 if (i == 2)
Breakpoint 3 at 0x400db753: file /home/user-name/esp/blink/main/./blink.c, line 34.
(gdb)
```

Above command sets conditional breakpoint to halt program execution in line `34` of `blink.c` if `i == 2`.

If current value of `i` is less than `2` and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt:

```
(gdb) set var i = 0
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB755 (active)    APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active)    APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB755 (active)    APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active)    APP_CPU: PC=0x400D112C

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:34
34          gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

Obtaining help on commands Commands presented so for should provide are very basis and intended to let you quickly get started with JTAG debugging. Check help what are the other commands at you disposal. To obtain help on syntax and functionality of particular command, being at `(gdb)` prompt type `help` and command name:

```
(gdb) help next
Step program, proceeding through subroutine calls.
Usage: next [N]
Unlike "step", if the current source line calls a subroutine,
this command does not enter the subroutine, but instead steps over
```

(continues on next page)

(continued from previous page)

```
the call, in effect treating it as a single source line.
(gdb)
```

By typing just `help`, you will get top level list of command classes, to aid you drilling down to more details. Optionally refer to available GDB cheat sheets, for instance https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf. Good to have as a reference (even if not all commands are applicable in an embedded environment).

Ending debugger session To quit debugger enter `q`:

```
(gdb) q
A debugging session is active.

    Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/user-name/esp/blink/build/blink.elf, Remote target
Ending remote debugging.
user-name@computer-name:~/esp/blink$
```

- *Using Debugger*
- *Debugging Examples*
- *Tips and Quirks*
- *Application Level Tracing library*
- Introduction to ESP-Prog Board

4.17 Linker Script Generation

4.17.1 Overview

There are several *memory regions* where code and data can be placed. Code and read-only data are placed by default in flash, writable data in RAM, etc. However, it is sometimes necessary to change these default placements.

For example, it may be necessary to place:

- critical code in RAM for performance reasons.
- executable code in IRAM so that it can be ran while cache is disabled.
- code in RTC memory for use in a wake stub.
- code in RTC memory for use by the ULP coprocessor.

With the linker script generation mechanism, it is possible to specify these placements at the component level within ESP-IDF. The component presents information on how it would like to place its symbols, objects or the entire archive. During build, the information presented by the components are collected, parsed and processed; and the placement rules generated is used to link the app.

4.17.2 Quick Start

This section presents a guide for quickly placing code/data to RAM and RTC memory - placements ESP-IDF provides out-of-the-box.

For this guide, suppose we have the following:

```
components
└── my_component
    ├── CMakeLists.txt
    ├── Kconfig
    ├── src/
    │   ├── my_src1.c
    │   ├── my_src2.c
    │   └── my_src3.c
    └── my_linker_fragment_file.lf
```

- a component named `my_component` that is archived as library `libmy_component.a` during build
- three source files archived under the library, `my_src1.c`, `my_src2.c` and `my_src3.c` which are compiled as `my_src1.o`, `my_src2.o` and `my_src3.o`, respectively
- under `my_src1.o`, the function `my_function1` is defined; under `my_src2.o`, the function `my_function2` is defined
- there is bool-type config `PERFORMANCE_MODE` (y/n) and int type config `PERFORMANCE_LEVEL` (with range 0-3) in `my_component`'s Kconfig

Creating and Specifying a Linker Fragment File

Before anything else, a linker fragment file needs to be created. A linker fragment file is simply a text file with a `.lf` extension upon which the desired placements will be written. After creating the file, it is then necessary to present it to the build system. The instructions for the build systems supported by ESP-IDF are as follows:

In the component's `CMakeLists.txt` file, specify argument `LDFRAGMENTS` in the `idf_component_register` call. The value of `LDFRAGMENTS` can either be an absolute path or a relative path from the component directory to the created linker fragment file.

```
# file paths relative to CMakeLists.txt
idf_component_register(...
                       LDFRAGMENTS "path/to/linker_fragment_file.lf" "path/to/
↪another_linker_fragment_file.lf"
                       ...
                       )
```

Specifying placements

It is possible to specify placements at the following levels of granularity:

- object file (`.obj` or `.o` files)
- symbol (function/variable)
- archive (`.a` files)

Placing object files Suppose the entirety of `my_src1.o` is performance-critical, so it is desirable to place it in RAM. On the other hand, the entirety of `my_src2.o` contains symbols needed coming out of deep sleep, so it needs to be put under RTC memory.

In the linker fragment file, we can write:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    my_src1 (noflash)      # places all my_src1 code/read-only data under IRAM/DRAM
    my_src2 (rtc)          # places all my_src2 code/ data and read-only data under␣
↪RTC fast memory/RTC slow memory
```

What happens to `my_src3.o`? Since it is not specified, default placements are used for `my_src3.o`. More on default placements *here*.

Placing symbols Continuing our example, suppose that among functions defined under `object1.o`, only `my_function1` is performance-critical; and under `object2.o`, only `my_function2` needs to execute after the chip comes out of deep sleep. This could be accomplished by writing:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    my_src1:my_function1 (noflash)
    my_src2:my_function2 (rtc)
```

The default placements are used for the rest of the functions in `my_src1.o` and `my_src2.o` and the entire `object3.o`. Something similar can be achieved for placing data by writing the variable name instead of the function name, like so:

```
my_src1:my_variable (noflash)
```

Warning: There are *limitations* in placing code/data at symbol granularity. In order to ensure proper placements, an alternative would be to group relevant code and data into source files, and *use object-granularity placements*.

Placing entire archive In this example, suppose that the entire component archive needs to be placed in RAM. This can be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    * (noflash)
```

Similarly, this places the entire component in RTC memory:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    * (rtc)
```

Configuration-dependent placements Suppose that the entire component library should only have special placement when a certain condition is true; for example, when `CONFIG_PERFORMANCE_MODE == y`. This could be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_MODE = y:
        * (noflash)
    else:
        * (default)
```

For a more complex config-dependent placement, suppose the following requirements: when `CONFIG_PERFORMANCE_LEVEL == 1`, only `object1.o` is put in RAM; when `CONFIG_PERFORMANCE_LEVEL == 2`, `object1.o` and `object2.o`; and when `CONFIG_PERFORMANCE_LEVEL == 3` all object files under the archive are to be put into RAM. When these three are false however, put entire library in RTC memory. This scenario is a bit contrived, but, it can be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_LEVEL = 1:
```

(continues on next page)

(continued from previous page)

```
        my_src1 (noflash)
    elif PERFORMANCE_LEVEL = 2:
        my_src1 (noflash)
        my_src2 (noflash)
    elif PERFORMANCE_LEVEL = 3:
        my_src1 (noflash)
        my_src2 (noflash)
        my_src3 (noflash)
    else:
        * (rtc)
```

Nesting condition-checking is also possible. The following is equivalent to the snippet above:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_LEVEL <= 3 && PERFORMANCE_LEVEL > 0:
        if PERFORMANCE_LEVEL >= 1:
            object1 (noflash)
            if PERFORMANCE_LEVEL >= 2:
                object2 (noflash)
                if PERFORMANCE_LEVEL >= 3:
                    object2 (noflash)
    else:
        * (rtc)
```

The 'default' placements

Up until this point, the term 'default placements' has been mentioned as fallback placements when the placement rules `rtc` and `noflash` are not specified. It is important to note that the tokens `noflash` or `rtc` are not merely keywords, but are actually entities called fragments, specifically *schemes*.

In the same manner as `rtc` and `noflash` are schemes, there exists a `default` scheme which defines what the default placement rules should be. As the name suggests, it is where code and data are usually placed, i.e. code/constants is placed in flash, variables placed in RAM, etc. More on the default scheme *here*.

Note: For an example of an ESP-IDF component using the linker script generation mechanism, see freertos/CMakeLists.txt. `freertos` uses this to place its object files to the instruction RAM for performance reasons.

This marks the end of the quick start guide. The following text discusses the internals of the mechanism in a little bit more detail. The following sections should be helpful in creating custom placements or modifying default behavior.

4.17.3 Linker Script Generation Internals

Linking is the last step in the process of turning C/C++ source files into an executable. It is performed by the toolchain's linker, and accepts linker scripts which specify code/data placements, among other things. With the linker script generation mechanism, this process is no different, except that the linker script passed to the linker is dynamically generated from: (1) the collected *linker fragment files* and (2) *linker script template*.

Note: The tool that implements the linker script generation mechanism lives under tools/ldgen.

Linker Fragment Files

As mentioned in the quick start guide, fragment files are simple text files with the `.lf` extension containing the desired placements. This is a simplified description of what fragment files contain, however. What fragment files actually contain are 'fragments'. Fragments are entities which contain pieces of information which, when put together, form placement rules that tell where to place sections of object files in the output binary. There are three types of fragments: *sections*, *scheme* and *mapping*.

Grammar The three fragment types share a common grammar:

```
[type:name]
key: value
key:
    value
    value
    value
    ...
```

- type: Corresponds to the fragment type, can either be `sections`, `scheme` or `mapping`.
- name: The name of the fragment, should be unique for the specified fragment type.
- key, value: Contents of the fragment; each fragment type may support different keys and different grammars for the key values.
  - For *sections* and *scheme*, the only supported key is `entries`
  - For *mappings*, both `archive` and `entries` are supported.

Note: In cases where multiple fragments of the same type and name are encountered, an exception is thrown.

Note: The only valid characters for fragment names and keys are alphanumeric characters and underscore.

Condition Checking

Condition checking enable the linker script generation to be configuration-aware. Depending on whether expressions involving configuration values are true or not, a particular set of values for a key can be used. The evaluation uses `eval_string` from kconfiglib package and adheres to its required syntax and limitations. Supported operators are as follows:

- **comparison**
  - LessThan <
  - LessThanOrEqualTo <=
  - MoreThan >
  - MoreThanOrEqualTo >=
  - Equal =
  - NotEqual !=
- **logical**
  - Or ||
  - And &&
  - Negation !
- **grouping**
  - Parenthesis ()

Condition checking behaves as you would expect an `if...elseif/elif...else` block in other languages. Condition-checking is possible for both key values and entire fragments. The two sample fragments below are equivalent:

```
# Value for keys is dependent on config
[type:name]
key_1:
```

(continues on next page)

(continued from previous page)

```
    if CONDITION = y:
        value_1
    else:
        value_2
key_2:
    if CONDITION = y:
        value_a
    else:
        value_b
```

```
# Entire fragment definition is dependent on config
if CONDITION = y:
    [type:name]
    key_1:
        value_1
    key_2:
        value_a
else:
    [type:name]
    key_1:
        value_2
    key_2:
        value_b
```

Comments

Comment in linker fragment files begin with `#`. Like in other languages, comment are used to provide helpful descriptions and documentation and are ignored during processing.

Types Sections

Sections fragments defines a list of object file sections that the GCC compiler emits. It may be a default section (e.g. `.text`, `.data`) or it may be user defined section through the `__attribute__` keyword.

The use of an optional '+' indicates the inclusion of the section in the list, as well as sections that start with it. This is the preferred method over listing both explicitly.

```
[sections:name]
entries:
    .section+
    .section
    ...
```

Example:

```
# Non-preferred
[sections:text]
entries:
    .text
    .text.*
    .literal
    .literal.*

# Preferred, equivalent to the one above
[sections:text]
entries:
    .text+              # means .text and .text.*
    .literal+           # means .literal and .literal.*
```

Scheme

Scheme fragments define what `target` a sections fragment is assigned to.

```
[scheme:name]
entries:
    sections -> target
    sections -> target
    ...
```

Example:

```
[scheme:noflash]
entries:
    text -> iram0_text          # the entries under the sections fragment named␣
↪text will go to iram0_text
    rodata -> dram0_data        # the entries under the sections fragment named␣
↪rodata will go to dram0_data
```

The `default` scheme

There exists a special scheme with the name `default`. This scheme is special because catch-all placement rules are generated from its entries. This means that, if one of its entries is `text -> flash_text`, the placement rule will be generated for the target `flash_text`.

```
*(.literal .literal.* .text .text.*)
```

These catch-all rules then effectively serve as fallback rules for those whose mappings were not specified.

The `default scheme` is defined in esp_system/app.lf. The `noflash` and `rtc` scheme fragments which are built-in schemes referenced in the quick start guide are also defined in this file.

Mapping

Mapping fragments define what scheme fragment to use for mappable entities, i.e. object files, function names, variable names, archives.

```
[mapping:name]
archive: archive                # output archive file name, as built (i.e. libxxx.
↪a)
entries:
    object:symbol (scheme)      # symbol granularity
    object (scheme)             # object granularity
    * (scheme)                  # archive granularity
```

There are three levels of placement granularity:

- symbol: The object file name and symbol name are specified. The symbol name can be a function name or a variable name.
- object: Only the object file name is specified.
- archive: `*` is specified, which is a short-hand for all the object files under the archive.

To know what an entry means, let us expand a sample object-granularity placement:

```
object (scheme)
```

Then expanding the scheme fragment from its entries definitions, we have:

```
object (sections -> target,
        sections -> target,
        ...)
```

Expanding the sections fragment with its entries definition:

```
object (.section,       # given this object file
        .section,       # put its sections listed here at this
        ... -> target, # target
```

(continues on next page)

(continued from previous page)

```
        .section,
        .section,       # same should be done for these sections
        ... -> target,

        ...)            # and so on
```

Example:

```
[mapping:map]
archive: libfreertos.a
entries:
    * (noflash)
```

Aside from the entity and scheme, flags can also be specified in an entry. The following flags are supported (note: <> = argument name, [] = optional):

1. ALIGN(<alignment>[, pre, post])
   Align the placement by the amount specified in `alignment`. Generates
2. SORT([<sort_by_first>, <sort_by_second>])
   Emits `SORT_BY_NAME`, `SORT_BY_ALIGNMENT`, `SORT_BY_INIT_PRIORITY` or `SORT` in the input section description.
   Possible values for `sort_by_first` and `sort_by_second` are: `name`, `alignment`, `init_priority`.
   If both `sort_by_first` and `sort_by_second` are not specified, the input sections are sorted by name. If both are specified, then the nested sorting follows the same rules discussed in https://sourceware.org/binutils/docs/ld/Input-Section-Wildcards.html.
3. KEEP()
   Prevent the linker from discarding the placement by surrounding the input section description with KEEP command. See https://sourceware.org/binutils/docs/ld/Input-Section-Keep.html for more details.

4.SURROUND(<name>)

Generate symbols before and after the placement. The generated symbols follow the naming `_<name>_start` and `_<name>_end`. For example, if `name` == sym1,

When adding flags, the specific `section -> target` in the scheme needs to be specified. For multiple `section -> target`, use a comma as a separator. For example,

```
# Notes:
# A. semicolon after entity-scheme
# B. comma before section2 -> target2
# C. section1 -> target1 and section2 -> target2 should be defined in entries of␣
↪scheme1
entity1 (scheme1);
    section1 -> target1 KEEP() ALIGN(4, pre, post),
    section2 -> target2 SURROUND(sym) ALIGN(4, post) SORT()
```

Putting it all together, the following mapping fragment, for example,

```
[mapping:name]
archive: lib1.a
entries:
    obj1 (noflash);
        rodata -> dram0_data KEEP() SORT() ALIGN(8) SURROUND(my_sym)
```

generates an output on the linker script:

```
. = ALIGN(8)
_my_sym_start = ABSOLUTE(.)
KEEP(lib1.a:obj1.*( SORT(.rodata) SORT(.rodata.*) ))
_my_sym_end = ABSOLUTE(.)
```

Note that ALIGN and SURROUND, as mentioned in the flag descriptions, are order sensitive. Therefore, if for the same mapping fragment these two are switched, the following is generated instead:

```
_my_sym_start = ABSOLUTE(.)
. = ALIGN(8)
KEEP(lib1.a:obj1.*( SORT(.rodata) SORT(.rodata.*) ))
_my_sym_end = ABSOLUTE(.)
```

On Symbol-Granularity Placements Symbol granularity placements is possible due to compiler flags `-ffunction-sections` and `-fdata-sections`. ESP-IDF compiles with these flags by default. If the user opts to remove these flags, then the symbol-granularity placements will not work. Furthermore, even with the presence of these flags, there are still other limitations to keep in mind due to the dependence on the compiler's emitted output sections.

For example, with `-ffunction-sections`, separate sections are emitted for each function; with section names predictably constructed i.e. `.text.{func_name}` and `.literal.{func_name}`. This is not the case for string literals within the function, as they go to pooled or generated section names.

With `-fdata-sections`, for global scope data the compiler predictably emits either `.data.{var_name}`, `.rodata.{var_name}` or `.bss.{var_name}`; and so `Type I` mapping entry works for these. However, this is not the case for static data declared in function scope, as the generated section name is a result of mangling the variable name with some other information.

Linker Script Template

The linker script template is the skeleton in which the generated placement rules are put into. It is an otherwise ordinary linker script, with a specific marker syntax that indicates where the generated placement rules are placed.

To reference the placement rules collected under a `target` token, the following syntax is used:

```
mapping[target]
```

Example:

The example below is an excerpt from a possible linker script template. It defines an output section `.iram0.text`, and inside is a marker referencing the target `iram0_text`.

```
.iram0.text :
{
    /* Code marked as runnning out of IRAM */
    _iram_text_start = ABSOLUTE(.);

    /* Marker referencing iram0_text */
    mapping[iram0_text]

    _iram_text_end = ABSOLUTE(.);
} > iram0_0_seg
```

Suppose the generator collected the fragment definitions below:

```
[sections:text]
    .text+
    .literal+
```

(continues on next page)

(continued from previous page)

```
[sections:iram]
    .iram1+

[scheme:default]
entries:
    text -> flash_text
    iram -> iram0_text

[scheme:noflash]
entries:
    text -> iram0_text

[mapping:freertos]
archive: libfreertos.a
entries:
    * (noflash)
```

Then the corresponding excerpt from the generated linker script will be as follows:

```
.iram0.text :
{
    /* Code marked as runnning out of IRAM */
    _iram_text_start = ABSOLUTE(.);

    /* Placement rules generated from the processed fragments, placed where the␣
↪marker was in the template */
    *(.iram1 .iram1.*)
    *libfreertos.a:(.literal .text .literal.* .text.*)

    _iram_text_end = ABSOLUTE(.);
} > iram0_0_seg
```

`*libfreertos.a:(.literal .text .literal.* .text.*)`

Rule generated from the entry `* (noflash)` of the `freertos` mapping fragment. All `text` sections of all object files under the archive `libfreertos.a` will be collected under the target `iram0_text` (as per the `noflash` scheme) and placed wherever in the template `iram0_text` is referenced by a marker.

`*(.iram1 .iram1.*)`

Rule generated from the default scheme entry `iram -> iram0_text`. Since the default scheme specifies an `iram -> iram0_text` entry, it too is placed wherever `iram0_text` is referenced by a marker. Since it is a rule generated from the default scheme, it comes first among all other rules collected under the same target name.

The linker script template currently used is esp_system/ld/esp32/sections.ld.in; the generated output script `sections.ld` is put under its build directory.

Migrate to ESP-IDF v5.0 Linker Script Fragment Files Grammar

The old grammar supported in ESP-IDF v3.x would be dropped in ESP-IDF v5.0. Here are a few notes on how to migrate properly:

1. Now indentation is enforced and improperly indented fragment files would generate a runtime parse exception. This was not enforced in the old version but previous documentation and examples demonstrate properly indented grammar.
2. Migrate the old condition entry to the `if...elif...else` structure for conditionals. You can refer to the *earlier chapter* for detailed grammar.
3. mapping fragments now requires a name like other fragment types.

4.18 lwIP

ESP-IDF uses the open source lwIP lightweight TCP/IP stack. The ESP-IDF version of lwIP (esp-lwip) has some modifications and additions compared to the upstream project.

4.18.1 Supported APIs

ESP-IDF supports the following lwIP TCP/IP stack functions:

- *BSD Sockets API*
- *Netconn API* is enabled but not officially supported for ESP-IDF applications

Adapted APIs

> **Warning:** When using any lwIP API (other than *BSD Sockets API*), please make sure that it is thread safe. To check if a given API call is safe, enable *CONFIG_LWIP_CHECK_THREAD_SAFETY* and run the application. This way lwIP asserts the TCP/IP core functionality to be correctly accessed; the execution aborts if it is not locked properly or accessed from the correct task (*lwIP FreeRTOS Task*). The general recommendation is to use *ESP-NETIF* component to interact with lwIP.

Some common lwIP "app" APIs are supported indirectly by ESP-IDF:

- DHCP Server & Client are supported indirectly via the *ESP-NETIF* functionality
- Domain Name System (DNS) is supported in lwIP; DNS servers could be assigned automatically when acquiring a DHCP address, or manually configured using the *ESP-NETIF* API.

Note: DNS server configuration in lwIP is global, not interface-specific. If you are using multiple network interfaces with distinct DNS servers, exercise caution to prevent inadvertent overwrites of one interface's DNS settings when acquiring a DHCP lease from another interface.

- Simple Network Time Protocol (SNTP) is supported via the lwip/include/apps/sntp/sntp.h lwip/lwip/src/include/lwip/apps/sntp.h functions (see also *SNTP Time Synchronization*)
- ICMP Ping is supported using a variation on the lwIP ping API. See *ICMP Echo*.
- NetBIOS lookup is available using the standard lwIP API. protocols/http_server/restful_server has an option to demonstrate using NetBIOS to look up a host on the LAN.
- mDNS uses a different implementation to the lwIP default mDNS (see *mDNS Service*), but lwIP can look up mDNS hosts using standard APIs such as `gethostbyname()` and the convention `hostname.local`, provided the *CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES* setting is enabled.

4.18.2 BSD Sockets API

The BSD Sockets API is a common cross-platform TCP/IP sockets API that originated in the Berkeley Standard Distribution of UNIX but is now standardized in a section of the POSIX specification. BSD Sockets are sometimes called POSIX Sockets or Berkeley Sockets.

As implemented in ESP-IDF, lwIP supports all of the common usages of the BSD Sockets API.

References

A wide range of BSD Sockets reference material is available, including:

- Single UNIX Specification BSD Sockets page
- Berkeley Sockets Wikipedia page

Examples

A number of ESP-IDF examples show how to use the BSD Sockets APIs:

- protocols/sockets/tcp_server
- protocols/sockets/tcp_client
- protocols/sockets/udp_server
- protocols/sockets/udp_client
- protocols/sockets/udp_multicast
- protocols/http_request (Note: this is a simplified example of using a TCP socket to send an HTTP request. The *ESP HTTP Client* is a much better option for sending HTTP requests.)

Supported functions

The following BSD socket API functions are supported. For full details see lwip/lwip/src/include/lwip/sockets.h.

- `socket()`
- `bind()`
- `accept()`
- `shutdown()`
- `getpeername()`
- `getsockopt()` & `setsockopt()` (see *Socket Options*)
- `close()` (via *Virtual filesystem component*)
- `read()`, `readv()`, `write()`, `writev()` (via *Virtual filesystem component*)
- `recv()`, `recvmsg()`, `recvfrom()`
- `send()`, `sendmsg()`, `sendto()`
- `select()` (via *Virtual filesystem component*)
- `poll()` (Note: on ESP-IDF, `poll()` is implemented by calling select internally, so using `select()` directly is recommended if a choice of methods is available.)
- `fcntl()` (see *fcntl*)

Non-standard functions:

- `ioctl()` (see *ioctls*)

Note: Some lwIP application sample code uses prefixed versions of BSD APIs, for example `lwip_socket()` instead of the standard `socket()`. Both forms can be used with ESP-IDF, but using standard names is recommended.

Socket Error Handling

BSD Socket error handling code is very important for robust socket applications. Normally the socket error handling involves the following aspects:

- Detecting the error.
- Geting the error reason code.
- Handle the error according to the reason code.

In lwIP, we have two different scenarios of handling socket errors:

- Socket API returns an error. For more information, see *Socket API Errors*.
- `select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout)` has exception descriptor indicating that the socket has an error. For more information, see *select() Errors*.

Socket API Errors

The error detection

- We can know that the socket API fails according to its return value.

Get the error reason code

- When socket API fails, the return value doesn't contain the failure reason and the application can get the error reason code by accessing errno. Different values indicate different meanings. For more information, see <*Socket Error Reason Code*>.

Example:

```
int err;
int sockfd;

if (sockfd = socket(AF_INET,SOCK_STREAM,0) < 0) {
    // the error code is obtained from errno
    err = errno;
    return err;
}
```

select() Errors

The error detection

- Socket error when `select()` has exception descriptor

Get the error reason code

- If the `select` indicates that the socket fails, we can't get the error reason code by accessing errno, instead we should call `getsockopt()` to get the failure reason code. Because `select()` has exception descriptor, the error code will not be given to errno.

Note: `getsockopt` function prototype `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)`. Its function is to get the current value of the option of any type, any state socket, and store the result in optval. For example, when you get the error code on a socket, you can get it by `getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &err, &optlen)`.

Example:

```
int err;

if (select(sockfd + 1, NULL, NULL, &exfds, &tval) <= 0) {
    err = errno;
    return err;
} else {
    if (FD_ISSET(sockfd, &exfds)) {
        // select() exception set using getsockopt()
        int optlen = sizeof(int);
        getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &err, &optlen);
        return err;
    }
}
```

Socket Error Reason Code Below is a list of common error codes. For more detailed list of standard POSIX/C error codes, please see newlib errno.h and the platform-specific extensions newlib/platform_include/errno.h

Error code	Description
ECONNREFUSED	Connection refused
EADDRINUSE	Address already in use
ECONNABORTED	Software caused connection abort
ENETUNREACH	Network is unreachable
ENETDOWN	Network interface is not configured
ETIMEDOUT	Connection timed out
EHOSTDOWN	Host is down
EHOSTUNREACH	Host is unreachable
EINPROGRESS	Connection already in progress
EALREADY	Socket already connected
EDESTADDRREQ	Destination address required
EPROTONOSUPPORT	Unknown protocol

Socket Options

The `getsockopt()` and `setsockopt()` functions allow getting/setting per-socket options.

Not all standard socket options are supported by lwIP in ESP-IDF. The following socket options are supported:

Common options Used with level argument `SOL_SOCKET`.

- `SO_REUSEADDR` (available if *CONFIG_LWIP_SO_REUSE* is set, behavior can be customized by setting *CONFIG_LWIP_SO_REUSE_RXTOALL*)
- `SO_KEEPALIVE`
- `SO_BROADCAST`
- `SO_ACCEPTCONN`
- `SO_RCVBUF` (available if *CONFIG_LWIP_SO_RCVBUF* is set)
- `SO_SNDTIMEO` / `SO_RCVTIMEO`
- `SO_ERROR` (this option is only used with `select()`, see *Socket Error Handling*)
- `SO_TYPE`
- `SO_NO_CHECK` (for UDP sockets only)

IP options Used with level argument `IPPROTO_IP`.

- `IP_TOS`
- `IP_TTL`
- `IP_PKTINFO` (available if *CONFIG_LWIP_NETBUF_RECVINFO* is set)

For multicast UDP sockets:

- `IP_MULTICAST_IF`
- `IP_MULTICAST_LOOP`
- `IP_MULTICAST_TTL`
- `IP_ADD_MEMBERSHIP`
- `IP_DROP_MEMBERSHIP`

TCP options TCP sockets only. Used with level argument `IPPROTO_TCP`.

- `TCP_NODELAY`

Options relating to TCP keepalive probes:

- `TCP_KEEPALIVE` (int value, TCP keepalive period in milliseconds)
- `TCP_KEEPIDLE` (same as `TCP_KEEPALIVE`, but the value is in seconds)
- `TCP_KEEPINTVL` (int value, interval between keepalive probes in seconds)
- `TCP_KEEPCNT` (int value, number of keepalive probes before timing out)

IPv6 options IPv6 sockets only. Used with level argument `IPPROTO_IPV6`

- `IPV6_CHECKSUM`
- `IPV6_V6ONLY`

For multicast IPv6 UDP sockets:

- `IPV6_JOIN_GROUP` / `IPV6_ADD_MEMBERSHIP`
- `IPV6_LEAVE_GROUP` / `IPV6_DROP_MEMBERSHIP`
- `IPV6_MULTICAST_IF`
- `IPV6_MULTICAST_HOPS`
- `IPV6_MULTICAST_LOOP`

fcntl

The `fcntl()` function is a standard API for manipulating options related to a file descriptor. In ESP-IDF, the *Virtual filesystem component* layer is used to implement this function.

When the file descriptor is a socket, only the following `fcntl()` values are supported:

- `O_NONBLOCK` to set/clear non-blocking I/O mode. Also supports `O_NDELAY`, which is identical to `O_NONBLOCK`.
- `O_RDONLY`, `O_WRONLY`, `O_RDWR` flags for different read/write modes. These can read via `F_GETFL` only, they cannot be set using `F_SETFL`. A TCP socket will return a different mode depending on whether the connection has been closed at either end or is still open at both ends. UDP sockets always return `O_RDWR`.

ioctls

The `ioctl()` function provides a semi-standard way to access some internal features of the TCP/IP stack. In ESP-IDF, the *Virtual filesystem component* layer is used to implement this function.

When the file descriptor is a socket, only the following `ioctl()` values are supported:

- `FIONREAD` returns the number of bytes of pending data already received in the socket's network buffer.
- `FIONBIO` is an alternative way to set/clear non-blocking I/O status for a socket, equivalent to `fcntl(fd, F_SETFL, O_NONBLOCK, ...)`.

4.18.3 Netconn API

lwIP supports two lower level APIs as well as the BSD Sockets API: the Netconn API and the Raw API.

The lwIP Raw API is designed for single threaded devices and is not supported in ESP-IDF.

The Netconn API is used to implement the BSD Sockets API inside lwIP, and it can also be called directly from ESP-IDF apps. This API has lower resource usage than the BSD Sockets API, in particular it can send and receive data without needing to first copy it into internal lwIP buffers.

Important: Espressif does not test the Netconn API in ESP-IDF. As such, this functionality is *enabled but not supported*. Some functionality may only work correctly when used from the BSD Sockets API.

For more information about the Netconn API, consult lwip/lwip/src/include/lwip/api.h and this wiki page which is part of the unofficial lwIP Application Developers Manual.

4.18.4 lwIP FreeRTOS Task

lwIP creates a dedicated TCP/IP FreeRTOS task to handle socket API requests from other tasks.

A number of configuration items are available to modify the task and the queues ("mailboxes") used to send data to/from the TCP/IP task:

- *CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*
- *CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*
- *CONFIG_LWIP_TCPIP_TASK_AFFINITY*

4.18.5 IPv6 Support

Both IPv4 and IPv6 are supported as dual stack and enabled by default (IPv6 may be disabled if it' s not needed, see *Minimum RAM usage*). IPv6 support is limited to *Stateless Autoconfiguration* only, *Stateful configuration* is not supported in ESP-IDF (not in upstream lwip). IPv6 Address configuration is defined by means of these protocols or services:

- **SLAAC** IPv6 Stateless Address Autoconfiguration (RFC-2462)
- **DHCPv6** Dynamic Host Configuration Protocol for IPv6 (RFC-8415)

None of these two types of address configuration is enabled by default, so the device uses only Link Local addresses or statically defined addresses.

Stateless Autoconfiguration Process

To enable address autoconfiguration using Router Advertisement protocol please enable:

- *CONFIG_LWIP_IPV6_AUTOCONFIG*

This configuration option enables IPv6 autoconfiguration for all network interfaces (in contrast to the upstream lwIP, where the autoconfiguration needs to be explicitly enabled for each netif with `netif->ip6_autoconfig_enabled=1`

DHCPv6

DHCPv6 in lwIP is very simple and support only stateless configuration. It could be enabled using:

- *CONFIG_LWIP_IPV6_DHCP6*

Since the DHCPv6 works only in its stateless configuration, the *Stateless Autoconfiguration Process* has to be enabled, too, by means of *CONFIG_LWIP_IPV6_AUTOCONFIG*. Moreover, the DHCPv6 needs to be explicitly enabled form the application code using

```
dhcp6_enable_stateless(netif);
```

DNS servers in IPv6 autoconfiguration

In order to autoconfigure DNS server(s), especially in IPv6 only networks, we have these two options

- Recursive domain name system –this belongs to the Neighbor Discovery Protocol (NDP), uses *Stateless Autoconfiguration Process*. Number of servers must be set *CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS*, this is option is disabled (set to 0) by default.
- DHCPv6 stateless configuration –uses *DHCPv6* to configure DNS servers. Note that the this configuration assumes IPv6 Router Advertisement Flags (RFC-5175) to be set to
  - Managed Address Configuration Flag = 0
  - Other Configuration Flag = 1

4.18.6 esp-lwip custom modifications

Additions

The following code is added which is not present in the upstream lwIP release:

Thread-safe sockets It is possible to `close()` a socket from a different thread to the one that created it. The `close()` call will block until any function calls currently using that socket from other tasks have returned.

It is, however, not possible to delete a task while it is actively waiting on `select()` or `poll()` APIs. It is always necessary that these APIs exit before destroying the task, as this might corrupt internal structures and cause subsequent crashes of the lwIP. (These APIs allocate globally referenced callback pointers on stack, so that when the task gets destroyed before unrolling the stack, the lwIP would still hold pointers to the deleted stack)

On demand timers lwIP IGMP and MLD6 features both initialize a timer in order to trigger timeout events at certain times.

The default lwIP implementation is to have these timers enabled all the time, even if no timeout events are active. This increases CPU usage and power consumption when using automatic light sleep mode. `esp-lwip` default behaviour is to set each timer "on demand" so it is only enabled when an event is pending.

To return to the default lwIP behaviour (always-on timers), disable *CONFIG_LWIP_TIMERS_ONDEMAND*.

Lwip timers API When users are not using WiFi, these APIs provide users with the ability to turn off LwIP timer to reduce power consumption.

The following API functions are supported. For full details see lwip/lwip/src/include/lwip/timeouts.h.

- `sys_timeouts_init()`
- `sys_timeouts_deinit()`

Additional Socket Options

- Some standard IPV4 and IPV6 multicast socket options are implemented (see *Socket Options*).
- Possible to set IPV6-only UDP and TCP sockets with `IPV6_V6ONLY` socket option (normal lwIP is TCP only).

IP layer features

- IPV4 source based routing implementation is different.
- IPV4 mapped IPV6 addresses are supported.

Customized lwIP hooks The original lwIP supports implementing custom compile-time modifications via `LWIP_HOOK_FILENAME`. This file is already used by the IDF port layer, but IDF users could still include and implement any custom additions via a header file defined by the macro `ESP_IDF_LWIP_HOOK_FILENAME`. Here is an exmaple of adding a custom hook file to the build process (the hook is called `my_hook.h` and located in the project's `main` folder):

```
idf_component_get_property(lwip lwip COMPONENT_LIB)
target_compile_options(${lwip} PRIVATE "-I${PROJECT_DIR}/main")
target_compile_definitions(${lwip} PRIVATE "-DESP_IDF_LWIP_HOOK_FILENAME=\"my_hook.
↪h\"")
```

Customized lwIP Options From ESP-IDF Build System The most common lwIP options are configurable through the component configuration menu. However, certain definitions need to be injected from the command line. The CMake function `target_compile_definitions()` can be employed to define macros, as illustrated below:

```
idf_component_get_property(lwip lwip COMPONENT_LIB)
target_compile_definitions(${lwip} PRIVATE "-DETHARP_SUPPORT_VLAN=1")
```

This approach may not work for function-like macros, as there is no guarantee that the definition will be accepted by all compilers, although it is supported in GCC. To address this limitation, the `add_definitions()` function can be utilized to define the macro for the entire project, for example: `add_definitions("-DFALLBACK_DNS_SERVER_ADDRESS(addr)=\"IP_ADDR4((addr), 8, 8,8,8)\"")`.

Alternatively, you can define your function-like macro in a header file which will be pre-included as an lwIP hook file, see *Customized lwIP hooks*.

Limitations

ESP-IDF additions to lwIP still suffer from the global DNS limitation, described in *Adapted APIs*. To address this limitation from application code, the `FALLBACK_DNS_SERVER_ADDRESS()` macro can be utilized to define a global DNS fallback server accessible from all interfaces. Alternatively, you have the option to maintain per-interface DNS servers and reconfigure them whenever the default interface changes.

Calling `send()` or `sendto()` repeatedly on a UDP socket may eventually fail with `errno` equal to `ENOMEM`. This is a limitation of buffer sizes in the lower layer network interface drivers. If all driver transmit buffers are full then UDP transmission will fail. Applications sending a high volume of UDP datagrams who don't wish for any to be dropped by the sender should check for this error code and re-send the datagram after a short delay.

Increasing the number of TX buffers in the *Wi-Fi* or *Ethernet* project configuration (as applicable) may also help.

4.18.7 Performance Optimization

TCP/IP performance is a complex subject, and performance can be optimized towards multiple goals. The default settings of ESP-IDF are tuned for a compromise between throughput, latency, and moderate memory usage.

Maximum throughput

Espressif tests ESP-IDF TCP/IP throughput using the wifi/iperf example in an RF sealed enclosure.

The wifi/iperf/sdkconfig.defaults file for the iperf example contains settings known to maximize TCP/IP throughput, usually at the expense of higher RAM usage. To get maximum TCP/IP throughput in an application at the expense of other factors then suggest applying settings from this file into the project sdkconfig.

Important: Suggest applying changes a few at a time and checking the performance each time with a particular application workload.

- If a lot of tasks are competing for CPU time on the system, consider that the lwIP task has configurable CPU affinity (*CONFIG_LWIP_TCPIP_TASK_AFFINITY*) and runs at fixed priority `ESP_TASK_TCPIP_PRIO` (18). Configure competing tasks to be pinned to a different core, or to run at a lower priority. See also *Built-In Task Priorities*.
- If using `select()` function with socket arguments only, disabling *CONFIG_VFS_SUPPORT_SELECT* will make `select()` calls faster.
- If there is enough free IRAM, select *CONFIG_LWIP_IRAM_OPTIMIZATION* to improve TX/RX throughput

If using a Wi-Fi network interface, please also refer to *Wi-Fi Buffer Usage*.

Minimum latency

Except for increasing buffer sizes, most changes which increase throughput will also decrease latency by reducing the amount of CPU time spent in lwIP functions.

- For TCP sockets, lwIP supports setting the standard `TCP_NODELAY` flag to disable Nagle's algorithm.

Minimum RAM usage

Most lwIP RAM usage is on-demand, as RAM is allocated from the heap as needed. Therefore, changing lwIP settings to reduce RAM usage may not change RAM usage at idle but can change it at peak.

- Reducing *CONFIG_LWIP_MAX_SOCKETS* reduces the maximum number of sockets in the system. This will also cause TCP sockets in the `WAIT_CLOSE` state to be closed and recycled more rapidly (if needed to open a new socket), further reducing peak RAM usage.
- Reducing *CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*, *CONFIG_LWIP_TCP_RECVMBOX_SIZE* and *CONFIG_LWIP_UDP_RECVMBOX_SIZE* reduce memory usage at the expense of throughput, depending on usage.
- Reducing *CONFIG_LWIP_TCP_MSL*, *CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT* reduces the maximum segment lifetime in the system. This will also cause TCP sockets in the `TIME_WAIT`, `FIN_WAIT_2` state to be closed and recycled more rapidly
- Disable *CONFIG_LWIP_IPV6* can save about 39 KB for firmware size and 2KB RAM when system power up and 7KB RAM when TCPIP stack running. If there is no requirement for supporting IPV6 then it can be disabled to save flash and RAM footprint.

If using Wi-Fi, please also refer to *Wi-Fi Buffer Usage*.

Peak Buffer Usage The peak heap memory that lwIP consumes is the **theoretically-maximum memory** that the lwIP driver consumes. Generally, the peak heap memory that lwIP consumes depends on:

- the memory required to create a UDP connection: lwip_udp_conn
- the memory required to create a TCP connection: lwip_tcp_conn
- the number of UDP connections that the application has: lwip_udp_con_num
- the number of TCP connections that the application has: lwip_tcp_con_num
- the TCP TX window size: lwip_tcp_tx_win_size
- the TCP RX window size: lwip_tcp_rx_win_size

So, the peak heap memory that the LwIP consumes can be calculated with the following formula:

lwip_dynamic_peek_memory = (lwip_udp_con_num * lwip_udp_conn) + (lwip_tcp_con_num * (lwip_tcp_tx_win_size + lwip_tcp_rx_win_size + lwip_tcp_conn))

Some TCP-based applications need only one TCP connection. However, they may choose to close this TCP connection and create a new one when an error (such as a sending failure) occurs. This may result in multiple TCP connections existing in the system simultaneously, because it may take a long time for a TCP connection to close, according to the TCP state machine (refer to RFC793).

4.19 Memory Types

ESP32 chip has multiple memory types and flexible memory mapping features. This section describes how ESP-IDF uses these features by default.

ESP-IDF distinguishes between instruction memory bus (IRAM, IROM, RTC FAST memory) and data memory bus (DRAM, DROM). Instruction memory is executable, and can only be read or written via 4-byte aligned words. Data memory is not executable and can be accessed via individual byte operations. For more information about the different memory buses consult the *ESP32 Technical Reference Manual* > *System and Memory* [PDF].

4.19.1 DRAM (Data RAM)

Non-constant static data (.data) and zero-initialized data (.bss) is placed by the linker into Internal SRAM as data memory. The remaining space in this region is used for the runtime heap.

By applying the `EXT_RAM_BSS_ATTR` macro, zero-initialized data can also be placed into external RAM. To use this macro, the *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY* needs to be enabled. See *Allow .bss Segment to be Placed in External Memory*.

The available size of the internal DRAM region is reduced by 64 KB (by shifting start address to `0x3FFC0000`) if Bluetooth stack is used. Length of this region is also reduced by 16 KB or 32 KB if trace memory is used. Due to some memory fragmentation issues caused by ROM, it is also not possible to use all available DRAM for static allocations - however the remaining DRAM is still available as heap at runtime.

Constant data may also be placed into DRAM, for example if it is used in an non-flash-safe ISR (see explanation under *How to place code in IRAM*).

"noinit" DRAM

The macro `__NOINIT_ATTR` can be used as attribute to place data into `.noinit` section. The values placed into this section will not be initialized at startup and should keep its value after software restart.

By applying the `EXT_RAM_NOINIT_ATTR` macro, non-initialized value could also be placed in external RAM. To do this, the *CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY* needs to be enabled. See *Allow .noinit Segment to be Placed in External Memory*. If the *CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY* is not enabled, `EXT_RAM_NOINIT_ATTR` will behave just as `__NOINIT_ATTR`, it will make data to be placed into `.noinit` segment in internal RAM.

Example:

```
__NOINIT_ATTR uint32_t noinit_data;
```

4.19.2 IRAM (Instruction RAM)

ESP-IDF allocates part of the Internal SRAM0 region for instruction RAM. The region is defined in *ESP32 Technical Reference Manual* > *System and Memory* > *Embedded Memory* [PDF]. Except for the first 64 KB block which is used for PRO and APP MMU caches, the rest of this memory range (i.e. from `0x40080000` to `0x400A0000`) is used to store parts of the application which need to run from RAM.

When to place code in IRAM

Cases when parts of the application should be placed into IRAM:

- Interrupt handlers must be placed into IRAM if `ESP_INTR_FLAG_IRAM` is used when registering the interrupt handler. For more information, see *IRAM-Safe Interrupt Handlers*.
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32 reads code and data from flash via the MMU cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss and significantly improve that function's performance.

How to place code in IRAM

Some code is automatically placed into the IRAM region using the linker script.

If some specific application code needs to be placed into IRAM, it can be done by using the *Linker Script Generation* feature and adding a linker script fragment file to your component that targets at the entire source files or functions with the `noflash` placement. See the *Linker Script Generation* docs for more information.

Alternatively, it's possible to specify IRAM placement in the source code using the `IRAM_ATTR` macro:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
        // ...
}
```

There are some possible issues with placement in IRAM, that may cause problems with IRAM-safe interrupt handlers:

- Strings or constants inside an IRAM_ATTR function may not be placed in RAM automatically. It's possible to use DRAM_ATTR attributes to mark these, or using the linker script method will cause these to be automatically placed correctly.

```
void IRAM_ATTR gpio_isr_handler(void* arg)
{
    const static DRAM_ATTR uint8_t INDEX_DATA[] = { 45, 33, 12, 0 };
    const static char *MSG = DRAM_STR("I am a string stored in RAM");
}
```

Note that knowing which data should be marked with DRAM_ATTR can be hard, the compiler will sometimes recognize that a variable or expression is constant (even if it is not marked const) and optimize it into flash, unless it is marked with DRAM_ATTR.

- GCC optimizations that automatically generate jump tables or switch/case lookup tables place these tables in flash. IDF by default builds all files with -fno-jump-tables -fno-tree-switch-conversion flags to avoid this.

Jump table optimizations can be re-enabled for individual source files that don't need to be placed in IRAM. For instructions on how to add the -fno-jump-tables -fno-tree-switch-conversion options when compiling individual source files, see *Controlling Component Compilation*.

4.19.3 IROM (code executed from flash)

If a function is not explicitly placed into *IRAM (Instruction RAM)* or RTC memory, it is placed into flash. As IRAM is limited, most of an application's binary code must be placed into IROM instead.

The mechanism by which Flash MMU is used to allow code execution from flash is described in *ESP32 Technical Reference Manual* > *Memory Management and Protection Units (MMU, MPU)* [PDF].

During *Application Startup Flow*, the bootloader (which runs from IRAM) configures the MMU flash cache to map the app's instruction code region to the instruction space. Flash accessed via the MMU is cached using some internal SRAM and accessing cached flash data is as fast as accessing other types of internal memory.

4.19.4 DROM (data stored in flash)

By default, constant data is placed by the linker into a region mapped to the MMU flash cache. This is the same as the *IROM (code executed from flash)* section, but is for read-only data not executable code.

The only constant data not placed into this memory type by default are literal constants which are embedded by the compiler into application code. These are placed as the surrounding function's executable instructions.

The DRAM_ATTR attribute can be used to force constants from DROM into the *DRAM (Data RAM)* section (see above).

4.19.5 RTC Slow memory

Global and static variables used by code which runs from RTC memory must be placed into RTC Slow memory. For example *deep sleep* variables can be placed here instead of RTC FAST memory, or code and variables accessed by the *ULP Coprocessor programming*.

The attribute macro named RTC_NOINIT_ATTR can be used to place data into this type of memory. The values placed into this section keep their value after waking from deep sleep.

Example:

```
RTC_NOINIT_ATTR uint32_t rtc_noinit_data;
```

4.19.6 RTC FAST memory

The same region of RTC FAST memory can be accessed as both instruction and data memory. Code which has to run after wake-up from deep sleep mode has to be placed into RTC memory. Please check detailed description in *deep sleep* documentation.

RTC FAST memory can only be accessed by the PRO CPU.

In single core mode, remaining RTC FAST memory is added to the heap unless the option *CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP* is disabled. This memory can be used interchangeably with *DRAM (Data RAM)*, but is slightly slower to access and not DMA capable.

4.19.7 DMA Capable Requirement

Most peripheral DMA controllers (e.g. SPI, sdmmc, etc.) have requirements that sending/receiving buffers should be placed in DRAM and word-aligned. We suggest to place DMA buffers in static variables rather than in the stack. Use macro `DMA_ATTR` to declare global/local static variables like:

```
DMA_ATTR uint8_t buffer[]="I want to send something";

void app_main()
{
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8 * sizeof(buffer),
    };
    spi_device_transmit(spi, &temp);
    // other stuff
}
```

Or:

```
void app_main()
{
    DMA_ATTR static uint8_t buffer[] = "I want to send something";
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8 * sizeof(buffer),
    };
    spi_device_transmit(spi, &temp);
    // other stuff
}
```

It is also possible to allocate DMA-capable memory buffers dynamically by using the *MALLOC_CAP_DMA* capabilities flag.

4.19.8 DMA Buffer in the stack

Placing DMA buffers in the stack is possible but discouraged. If doing so, pay attention to the following:

- Placing DRAM buffers on the stack is not recommended if the stack may be in PSRAM. If the stack of a task is placed in the PSRAM, several steps have to be taken as described in *Support for External RAM*.
- Use macro `WORD_ALIGNED_ATTR` in functions before variables to place them in proper positions like:

```
void app_main()
{
    uint8_t stuff;
    WORD_ALIGNED_ATTR uint8_t buffer[] = "I want to send something";    //or␣
↪the buffer will be placed right after stuff.
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8 * sizeof(buffer),
    };
    spi_device_transmit(spi, &temp);
    // other stuff
}
```

4.20 OpenThread

OpenThread is an IP stack running on the 802.15.4 MAC layer which features mesh network and low power consumption.

4.20.1 Modes of the OpenThread stack

OpenThread can run under the following modes on Espressif chips:

Standalone Node

The full OpenThread stack and the application layer run on the same chip. This mode is available on chips with 15.4 radio such as ESP32-H2 and ESP32-C6.

Radio Co-Processor (RCP)

The chip is connected to another host running the OpenThread IP stack. It sends and receives 15.4 packets on behalf of the host. This mode is available on chips with 15.4 radio such as ESP32-H2 and ESP32-C6. The underlying transport between the chip and the host can be SPI or UART. For the sake of latency, we recommend using SPI as the underlying transport.

OpenThread Host

For chips without a 15.4 radio, it can be connected to an RCP and run OpenThread under host mode. This mode enables OpenThread on Wi-Fi chips such as ESP32, ESP32-S2, ESP32-S3, and ESP32-C3. The following diagram shows how devices work under different modes:

4.20.2 How to Write an OpenThread Application

The OpenThread openthread/ot_cli example is a good place to start at. It demonstrates basic OpenThread initialization and simple socket-based server and client.

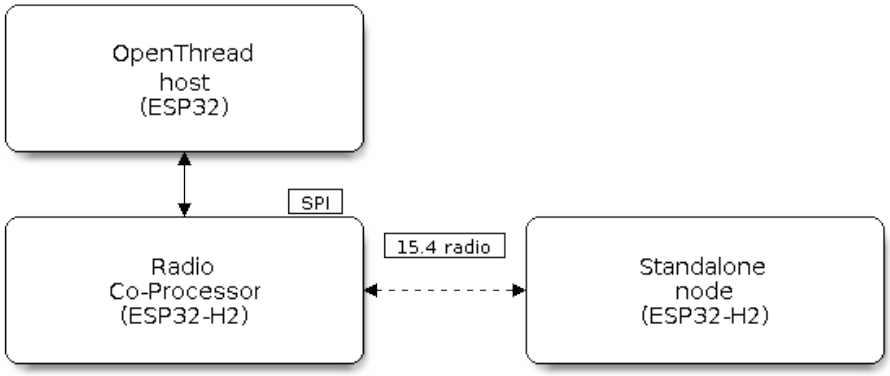


Fig. 54: OpenThread device modes

Before OpenThread Initialization

- s1.1: The main task calls `esp_vfs_eventfd_register()` to initialize the eventfd virtual file system. The eventfd file system is used for task notification in the OpenThread driver.
- s1.2: The main task calls `nvs_flash_init()` to initialize the NVS where the Thread network data is stored.
- s1.3: **Optional**. The main task calls `esp_netif_init()` only when it wants to create the network interface for Thread.
- s1.4: The main task calls `esp_event_loop_create()` to create the system Event task and initialize an application event' s callback function.

OpenThread Stack Initialization

- s2.1: Call `esp_openthread_init()` to initialize the OpenThread stack.

OpenThread Network Interface Initialization

The whole stage is **optional** and only required if the application wants to create the network interface for Thread.

- s3.1: Call `esp_netif_new()` with `ESP_NETIF_DEFAULT_OPENTHREAD` to create the interface.
- s3.2: Call `esp_openthread_netif_glue_init()` to create the OpenThread interface handlers.
- s3.3: Call `esp_netif_attach()` to attach the handlers to the interface.

The OpenThread Main Loop

- s4.3: Call `esp_openthread_launch_mainloop()` to launch the OpenThread main loop. Note that this is a busy loop and does not return until the OpenThread stack is terminated.

Calling OpenThread APIs

The OpenThread APIs are not thread-safe. When calling OpenThread APIs from other tasks, make sure to hold the lock with `esp_openthread_lock_acquire()` and release the lock with `esp_openthread_lock_release()` afterwards.

Deinitialization

The following steps are required to deinitialize the OpenThread stack:

- Call *esp_netif_destroy()* and *esp_openthread_netif_glue_deinit()* to deinitialize the OpenThread network interface if you have created one.
- Call *esp_openthread_deinit()* to deinitialize the OpenThread stack.

4.20.3 The OpenThread Border Router

The OpenThread border router connects the Thread network with other IP networks. It provides IPv6 connectivity, service registration, and commission functionality.

To launch an OpenThread border router on an ESP chip, you need to connect an RCP to a Wi-Fi capable chip such as ESP32.

Calling *esp_openthread_border_router_init()* during the initialization launches all the border routing functionalities.

You may refer to the openthread/ot_br example and the README for further border router details.

4.21 Partition Tables

4.21.1 Overview

A single ESP32 ' s flash can contain multiple apps, as well as many different kinds of data (calibration data, filesystems, parameter storage, etc). For this reason a partition table is flashed to (*default offset*) 0x8000 in the flash.

Partition table length is 0xC00 bytes (maximum 95 partition table entries). An MD5 checksum, which is used for checking the integrity of the partition table, is appended after the table data.

Each entry in the partition table has a name (label), type (app, data, or something else), subtype and the offset in flash where the partition is loaded.

The simplest way to use the partition table is to open the project configuration menu (`idf.py menuconfig`) and choose one of the simple predefined partition tables under *CONFIG_PARTITION_TABLE_TYPE*:

- "Single factory app, no OTA"
- "Factory app, two OTA definitions"

In both cases the factory app is flashed at offset 0x10000. If you execute *idf.py partition-table* then it will print a summary of the partition table.

4.21.2 Built-in Partition Tables

Here is the summary printed for the "Single factory app, no OTA" configuration:

```
# ESP-IDF Partition Table
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 1M,
```

- At a 0x10000 (64 KB) offset in the flash is the app labelled "factory". The bootloader will run this app by default.
- There are also two data regions defined in the partition table for storing NVS library partition and PHY init data.

Here is the summary printed for the "Factory app, two OTA definitions" configuration:

```
# ESP-IDF Partition Table
# Name,    Type, SubType, Offset,  Size, Flags
nvs,       data, nvs,     0x9000,  0x4000,
otadata,   data, ota,     0xd000,  0x2000,
phy_init,  data, phy,     0xf000,  0x1000,
factory,   app,  factory, 0x10000,  1M,
ota_0,     app,  ota_0,   0x110000, 1M,
ota_1,     app,  ota_1,   0x210000, 1M,
```

- There are now three app partition definitions. The type of the factory app (at 0x10000) and the next two "OTA" apps are all set to "app", but their subtypes are different.
- There is also a new "otadata" slot, which holds the data for OTA updates. The bootloader consults this data in order to know which app to execute. If "ota data" is empty, it will execute the factory app.

4.21.3 Creating Custom Tables

If you choose "Custom partition table CSV" in menuconfig then you can also enter the name of a CSV file (in the project directory) to use for your partition table. The CSV file can describe any number of definitions for the table you need.

The CSV format is the same format as printed in the summaries shown above. However, not all fields are required in the CSV. For example, here is the "input" CSV for the OTA partition table:

```
# Name,    Type, SubType,  Offset,   Size,  Flags
nvs,       data, nvs,      0x9000,  0x4000
otadata,   data, ota,      0xd000,  0x2000
phy_init,  data, phy,      0xf000,  0x1000
factory,   app,  factory,  0x10000,  1M
ota_0,     app,  ota_0,    ,         1M
ota_1,     app,  ota_1,    ,         1M
nvs_key,   data, nvs_keys, ,        0x1000
```

- Whitespace between fields is ignored, and so is any line starting with # (comments).
- Each non-comment line in the CSV file is a partition definition.
- The "Offset" field for each partition is empty. The gen_esp32part.py tool fills in each blank offset, starting after the partition table and making sure each partition is aligned correctly.

Name field

Name field can be any meaningful name. It is not significant to the ESP32. Names longer than 16 characters will be truncated.

Type field

Partition type field can be specified as `app` (0x00) or `data` (0x01). Or it can be a number 0-254 (or as hex 0x00-0xFE). Types 0x00-0x3F are reserved for ESP-IDF core functions.

If your app needs to store data in a format not already supported by ESP-IDF, then please add a custom partition type value in the range 0x40-0xFE.

See *`esp_partition_type_t`* for the enum definitions for `app` and `data` partitions.

If writing in C++ then specifying a application-defined partition type requires casting an integer to *`esp_partition_type_t`* in order to use it with the *partition API*. For example:

```
static const esp_partition_type_t APP_PARTITION_TYPE_A = (esp_partition_type_
↪t)0x40;
```

The ESP-IDF bootloader ignores any partition types other than `app` (0x00) and `data` (0x01).

SubType

The 8-bit subtype field is specific to a given partition type. ESP-IDF currently only specifies the meaning of the subtype field for `app` and `data` partition types.

See enum *`esp_partition_subtype_t`* for the full list of subtypes defined by ESP-IDF, including the following:

- When type is `app`, the subtype field can be specified as `factory` (0x00), `ota_0` (0x10) ⋯`ota_15` (0x1F) or `test` (0x20).
  - `factory` (0x00) is the default app partition. The bootloader will execute the factory app unless there it sees a partition of type data/ota, in which case it reads this partition to determine which OTA image to boot.
    - OTA never updates the factory partition.
    - If you want to conserve flash usage in an OTA project, you can remove the factory partition and use `ota_0` instead.
  - `ota_0` (0x10) ⋯`ota_15` (0x1F) are the OTA app slots. When *OTA* is in use, the OTA data partition configures which app slot the bootloader should boot. When using OTA, an application should have at least two OTA application slots (`ota_0` & `ota_1`). Refer to the *OTA documentation* for more details.
  - `test` (0x20) is a reserved subtype for factory test procedures. It will be used as the fallback boot partition if no other valid app partition is found. It is also possible to configure the bootloader to read a GPIO input during each boot, and boot this partition if the GPIO is held low, see *Boot from Test Firmware*.
- When type is `data`, the subtype field can be specified as `ota` (0x00), `phy` (0x01), `nvs` (0x02), nvs_keys (0x04), or a range of other component-specific subtypes (see *`subtype enum`*).
  - `ota` (0) is the *OTA data partition* which stores information about the currently selected OTA app slot. This partition should be 0x2000 bytes in size. Refer to the *OTA documentation* for more details.
  - `phy` (1) is for storing PHY initialisation data. This allows PHY to be configured per-device, instead of in firmware.
    - In the default configuration, the phy partition is not used and PHY initialisation data is compiled into the app itself. As such, this partition can be removed from the partition table to save space.
    - To load PHY data from this partition, open the project configuration menu (`idf.py menuconfig`) and enable *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION* option. You will also need to flash your devices with phy init data as the esp-idf build system does not do this automatically.
  - `nvs` (2) is for the *Non-Volatile Storage (NVS) API*.
    - NVS is used to store per-device PHY calibration data (different to initialisation data).
    - NVS is used to store WiFi data if the *esp_wifi_set_storage(WIFI_STORAGE_FLASH)* initialisation function is used.
    - The NVS API can also be used for other application data.
    - It is strongly recommended that you include an NVS partition of at least 0x3000 bytes in your project.
    - If using NVS API to store a lot of data, increase the NVS partition size from the default 0x6000 bytes.
  - `nvs_keys` (4) is for the NVS key partition. See *Non-Volatile Storage (NVS) API* for more details.
    - It is used to store NVS encryption keys when *NVS Encryption* feature is enabled.
    - The size of this partition should be 4096 bytes (minimum partition size).
  - There are other predefined data subtypes for data storage supported by ESP-IDF. These include *FAT filesystem* (*`ESP_PARTITION_SUBTYPE_DATA_FAT`*), *SPIFFS* (*`ESP_PARTITION_SUBTYPE_DATA_SPIFFS`*), etc.

  Other subtypes of `data` type are reserved for future ESP-IDF uses.
- If the partition type is any application-defined value (range 0x40-0xFE), then `subtype` field can be any value chosen by the application (range 0x00-0xFE).
  Note that when writing in C++, an application-defined subtype value requires casting to type *`esp_partition_subtype_t`* in order to use it with the *partition API*.

Extra Partition SubTypes

A component can define a new partition subtype by setting the `EXTRA_PARTITION_SUBTYPES` property. This property is a CMake list, each entry of which is a comma separated string with `<type>,`

<subtype>, <value> format. The build system uses this property to add extra subtypes and creates fields named ESP_PARTITION_SUBTYPE_<type>_<subtype> in *esp_partition_type_t*. The project can use this subtype to define partitions in the partitions table CSV file and use the new fields in *esp_partition_type_t*.

Offset & Size

Partitions with blank offsets in the CSV file will start after the previous partition, or after the partition table in the case of the first partition.

Partitions of type app have to be placed at offsets aligned to 0x10000 (64K). If you leave the offset field blank, gen_esp32part.py will automatically align the partition. If you specify an unaligned offset for an app partition, the tool will return an error.

Sizes and offsets can be specified as decimal numbers, hex numbers with the prefix 0x, or size multipliers K or M (1024 and 1024*1024 bytes).

If you want the partitions in the partition table to work relative to any placement (*CONFIG_PARTITION_TABLE_OFFSET*) of the table itself, leave the offset field (in CSV file) for all partitions blank. Similarly, if changing the partition table offset then be aware that all blank partition offsets may change to match, and that any fixed offsets may now collide with the partition table (causing an error).

Flags

Only one flag is currently supported, encrypted. If this field is set to encrypted, this partition will be encrypted if *Flash Encryption* is enabled.

Note: app type partitions will always be encrypted, regardless of whether this flag is set or not.

4.21.4 Generating Binary Partition Table

The partition table which is flashed to the ESP32 is in a binary format, not CSV. The tool partition_table/gen_esp32part.py is used to convert between CSV and binary formats.

If you configure the partition table CSV name in the project configuration (idf.py menuconfig) and then build the project or run idf.py partition-table, this conversion is done as part of the build process.

To convert CSV to Binary manually:

```
python gen_esp32part.py input_partitions.csv binary_partitions.bin
```

To convert binary format back to CSV manually:

```
python gen_esp32part.py binary_partitions.bin input_partitions.csv
```

To display the contents of a binary partition table on stdout (this is how the summaries displayed when running idf.py partition-table are generated:

```
python gen_esp32part.py binary_partitions.bin
```

4.21.5 Partition Size Checks

The ESP-IDF build system will automatically check if generated binaries fit in the available partition space, and will fail with an error if a binary is too large.

Currently these checks are performed for the following binaries:

- Bootloader binary must fit in space before partition table (see *Bootloader Size*).
- App binary should fit in at least one partition of type "app". If the app binary doesn't fit in any app partition, the build will fail. If it only fits in some of the app partitions, a warning is printed about this.

Note: Although the build process will fail if the size check returns an error, the binary files are still generated and can be flashed (although they may not work if they are too large for the available space.)

MD5 checksum

The binary format of the partition table contains an MD5 checksum computed based on the partition table. This checksum is used for checking the integrity of the partition table during the boot.

The MD5 checksum generation can be disabled by the `--disable-md5sum` option of `gen_esp32part.py` or by the *CONFIG_PARTITION_TABLE_MD5* option. This is useful for example when one *uses a bootloader from ESP-IDF before v3.1* which cannot process MD5 checksums and the boot fails with the error message `invalid magic number 0xebeb`.

4.21.6 Flashing the partition table

- `idf.py partition-table-flash`: will flash the partition table with esptool.py.
- `idf.py flash`: Will flash everything including the partition table.

A manual flashing command is also printed as part of `idf.py partition-table` output.

Note: Note that updating the partition table doesn't erase data that may have been stored according to the old partition table. You can use `idf.py erase-flash` (or `esptool.py erase_flash`) to erase the entire flash contents.

4.21.7 Partition Tool (parttool.py)

The component *partition_table* provides a tool parttool.py for performing partition-related operations on a target device. The following operations can be performed using the tool:

- reading a partition and saving the contents to a file (read_partition)
- writing the contents of a file to a partition (write_partition)
- erasing a partition (erase_partition)
- retrieving info such as name, offset, size and flag ("encrypted") of a given partition (get_partition_info)

The tool can either be imported and used from another Python script or invoked from shell script for users wanting to perform operation programmatically. This is facilitated by the tool's Python API and command-line interface, respectively.

Python API

Before anything else, make sure that the *parttool* module is imported.

```
import sys
import os

idf_path = os.environ["IDF_PATH"]  # get value of IDF_PATH from environment
parttool_dir = os.path.join(idf_path, "components", "partition_table")  # parttool.
↪py lives in $IDF_PATH/components/partition_table
```

(continues on next page)

(continued from previous page)

```
sys.path.append(parttool_dir)  # this enables Python to find parttool module
from parttool import *  # import all names inside parttool module
```

The starting point for using the tool's Python API to do is create a *ParttoolTarget* object:

```
# Create a partool.py target device connected on serial port /dev/ttyUSB1
target = ParttoolTarget("/dev/ttyUSB1")
```

The created object can now be used to perform operations on the target device:

```
# Erase partition with name 'storage'
target.erase_partition(PartitionName("storage"))

# Read partition with type 'data' and subtype 'spiffs' and save to file 'spiffs.bin
↪'
target.read_partition(PartitionType("data", "spiffs"), "spiffs.bin")

# Write to partition 'factory' the contents of a file named 'factory.bin'
target.write_partition(PartitionName("factory"), "factory.bin")

# Print the size of default boot partition
storage = target.get_partition_info(PARTITION_BOOT_DEFAULT)
print(storage.size)
```

The partition to operate on is specified using *PartitionName* or *PartitionType* or PARTITION_BOOT_DEFAULT. As the name implies, these can be used to refer to partitions of a particular name, type-subtype combination, or the default boot partition.

More information on the Python API is available in the docstrings for the tool.

Command-line Interface

The command-line interface of *parttool.py* has the following structure:

```
parttool.py [command-args] [subcommand] [subcommand-args]

- command-args - These are arguments that are needed for executing the main␣
↪command (parttool.py), mostly pertaining to the target device
- subcommand - This is the operation to be performed
- subcommand-args - These are arguments that are specific to the chosen operation
```

```
# Erase partition with name 'storage'
parttool.py --port "/dev/ttyUSB1" erase_partition --partition-name=storage

# Read partition with type 'data' and subtype 'spiffs' and save to file 'spiffs.bin
↪'
parttool.py --port "/dev/ttyUSB1" read_partition --partition-type=data --partition-
↪subtype=spiffs --output "spiffs.bin"

# Write to partition 'factory' the contents of a file named 'factory.bin'
parttool.py --port "/dev/ttyUSB1" write_partition --partition-name=factory --input
↪"factory.bin"

# Print the size of default boot partition
parttool.py --port "/dev/ttyUSB1" get_partition_info --partition-boot-default --
↪info size
```

More information can be obtained by specifying *–help* as argument:

```
# Display possible subcommands and show main command argument descriptions
parttool.py --help

# Show descriptions for specific subcommand arguments
parttool.py [subcommand] --help
```

4.22 Performance

ESP-IDF ships with default settings that are designed for a trade-off between performance, resource usage, and available functionality.

These guides describe how to optimize a firmware application for a particular aspect of performance. Usually this involves some trade-off in terms of limiting available functions, or swapping one aspect of performance (such as execution speed) for another (such as RAM usage).

4.22.1 How to Optimize Performance

1. Decide what the performance-critical aspects of your application are (for example: a particular response time to a certain network operation, a particular startup time limit, particular peripheral data throughput, etc.).
2. Find a way to measure this performance (some methods are outlined in the guides below).
3. Modify the code and project configuration and compare the new measurement to the old measurement.
4. Repeat step 3 until the performance meets the requirements set out in step 1.

4.22.2 Guides

Maximizing Execution Speed

Overview Optimizing execution speed is a key element of software performance. Code that executes faster can also have other positive effects, like reducing overall power consumption. However, improving execution speed may have trade-offs with other aspects of performance such as *Minimizing Binary Size*.

Choose What To Optimize If a function in the application firmware is executed once per week in the background, it may not matter if that function takes 10 ms or 100 ms to execute. If a function is executed constantly at 10 Hz, it matters greatly if it takes 10 ms or 100 ms to execute.

Most application firmwares will only have a small set of functions which require optimal performance. Perhaps those functions are executed very often, or have to meet some application requirements for latency or throughput. Optimization efforts should be targeted at these particular functions.

Measuring Performance The first step to improving something is to measure it.

Basic Performance Measurements If measuring performance relative to an external interaction with the world, you may be able to measure this directly (for example see the examples wifi/iperf and ethernet/iperf for measuring general network performance, or you can use an oscilloscope or logic analyzer to measure timing of an interaction with a device peripheral.)

Otherwise, one way to measure performance is to augment the code to take timing measurements:

```
#include "esp_timer.h"

void measure_important_function(void) {
    const unsigned MEASUREMENTS = 5000;
    uint64_t start = esp_timer_get_time();
```

(continues on next page)

(continued from previous page)

```
    for (int retries = 0; retries < MEASUREMENTS; retries++) {
        important_function(); // This is the thing you need to measure
    }

    uint64_t end = esp_timer_get_time();

    printf("%u iterations took %llu milliseconds (%llu microseconds per␣
↪invocation)\n",
           MEASUREMENTS, (end - start)/1000, (end - start)/MEASUREMENTS);
}
```

Executing the target multiple times can help average out factors like RTOS context switches, overhead of measurements, etc.

- Using *esp_timer_get_time()* generates "wall clock" timestamps with microsecond precision, but has moderate overhead each time the timing functions are called.
- It's also possible to use the standard Unix `gettimeofday()` and `utime()` functions, although the overhead is slightly higher.
- Otherwise, including `hal/cpu_hal.h` and calling the HAL function `cpu_hal_get_cycle_count()` will return the number of CPU cycles executed. This function has lower overhead than the others. It is good for measuring very short execution times with high precision.
  The CPU cycles are counted per-core, so only use this method from an interrupt handler, or a task that is pinned to a single core.
- If making "microbenchmarks" (i.e. benchmarking only a very small routine of code that runs in less than 1-2 milliseconds) then flash cache performance can sometimes cause big variations in timing measurements depending on the binary. This happens because binary layout can cause different patterns of cache misses in a particular sequence of execution. If the test code is larger then this effect usually averages out. Executing a small function multiple times when benchmarking can help reduce the impact of flash cache misses. Alternatively, move this code to IRAM (see *Targeted Optimizations*).

External Tracing The *Application Level Tracing library* allows measuring code execution with minimal impact on the code itself.

Tasks If the option *CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS* is enabled then the FreeRTOS API *vTaskGetRunTimeStats()* can be used to retrieve runtime information about the processor time used by each FreeRTOS task.

SEGGER SystemView is an excellent tool for visualizing task execution and looking for performance issues or improvements in the system as a whole.

Improving Overall Speed The following optimizations will improve the execution of nearly all code - including boot times, throughput, latency, etc:

- Set *CONFIG_ESPTOOLPY_FLASHFREQ* to 80 MHz. This is double the 40 MHz default value and will double the speed at which code is loaded or executed from flash. You should verify that the board or module that connects the ESP32 to the flash chip is rated for 80 MHz operation at the relevant temperature ranges, before changing this setting. The hardware datasheet(s) will have this information.
- Set *CONFIG_ESPTOOLPY_FLASHMODE* to QIO or QOUT mode (Quad I/O). Both will almost double the speed at which code is loaded or executed from flash compared to the default DIO mode. QIO is slightly faster than QOUT if both are supported. Note that both the flash chip model and the electrical connections between the ESP32 and the flash chip must support quad I/O modes or the SoC will not work correctly.
- Set *CONFIG_COMPILER_OPTIMIZATION* to "Optimize for performance (-O2)". This may slightly increase binary size compared to the default setting, but will almost certainly increase performance of some code. Note that if your code contains C or C++ Undefined Behaviour then increasing the compiler optimization level may expose bugs that otherwise are not seen.

- If the application uses PSRAM and is based on ESP32 rev. 3 (ECO3), setting *CONFIG_ESP32_REV_MIN* to `3` will disable PSRAM bug workarounds, reducing the code size and improving overall performance.
- Avoid using floating point arithmetic (`float`). Even though ESP32 has a single precision hardware floating point unit, floating point calculations are always slower than integer calculations. If possible then use fixed point representations, a different method of integer representation, or convert part of the calculation to be integer only before switching to floating point.
- Avoid using double precision floating point arithmetic (`double`). These calculations are emulated in software and are very slow. If possible then use an integer-based representation, or single-precision floating point.

Reduce Logging Overhead Although standard output is buffered, it's possible for an application to be limited by the rate at which it can print data to log output once buffers are full. This is particularly relevant for startup time if a lot of output is logged, but can happen at other times as well. There are multiple ways to solve this problem:

- Reduce the volume of log output by lowering the app *CONFIG_LOG_DEFAULT_LEVEL* (the equivalent bootloader setting is *CONFIG_BOOTLOADER_LOG_LEVEL*). This also reduces the binary size, and saves some CPU time spent on string formatting.
- Increase the speed of logging output by increasing the *CONFIG_ESP_CONSOLE_UART_BAUDRATE*

Not Recommended The following options will also increase execution speed, but are not recommended as they also reduce the debuggability of the firmware application and may increase the severity of any bugs.

- Set *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL* to disabled. This also reduces firmware binary size by a small amount. However, it may increase the severity of bugs in the firmware including security-related bugs. If necessary to do this to optimize a particular function, consider adding `#define NDEBUG` in the top of that single source file instead.

Targeted Optimizations The following changes will increase the speed of a chosen part of the firmware application:

- Move frequently executed code to IRAM. By default, all code in the app is executed from flash cache. This means that it's possible for the CPU to have to wait on a "cache miss" while the next instructions are loaded from flash. Functions which are copied into IRAM are loaded once at boot time, and then will always execute at full speed.
  IRAM is a limited resource, and using more IRAM may reduce available DRAM, so a strategic approach is needed when moving code to IRAM. See *IRAM (Instruction RAM)* for more information.
- Jump table optimizations can be re-enabled for individual source files that don't need to be placed in IRAM. For hot paths in large switch cases this will improve performance. For instructions on how to add the -fjump-tables -ftree-switch-conversion options when compiling individual source files, see *Controlling Component Compilation*

Improving Startup Time In addition to the overall performance improvements shown above, the following options can be tweaked to specifically reduce startup time:

- Minimizing the *CONFIG_LOG_DEFAULT_LEVEL* and *CONFIG_BOOTLOADER_LOG_LEVEL* has a large impact on startup time. To enable more logging after the app starts up, set the *CONFIG_LOG_MAXIMUM_LEVEL* as well and then call `esp_log_level_set()` to restore higher level logs. The system/startup_time main function shows how to do this.
- If using deep sleep, setting *CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP* allows a faster wake from sleep. Note that if using Secure Boot this represents a security compromise, as Secure Boot validation will not be performed on wake.

- Setting *CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON* will skip verifying the binary on every boot from power-on reset. How much time this saves depends on the binary size and the flash settings. Note that this setting carries some risk if the flash becomes corrupt unexpectedly. Read the help text of the *config item* for an explanation and recommendations if using this option.
- It's possible to save a small amount of time during boot by disabling RTC slow clock calibration. To do so, set *CONFIG_RTC_CLK_CAL_CYCLES* to 0. Any part of the firmware that uses RTC slow clock as a timing source will be less accurate as a result.

The example project system/startup_time is pre-configured to optimize startup time. The file system/startup_time/sdkconfig.defaults contain all of these settings. You can append these to the end of your project's own `sdkconfig` file to merge the settings, but please read the documentation for each setting first.

Task Priorities As ESP-IDF FreeRTOS is a real-time operating system, it's necessary to ensure that high throughput or low latency tasks are granted a high priority in order to run immediately. Priority is set when calling *`xTaskCreate()`* or *`xTaskCreatePinnedToCore()`* and can be changed at runtime by calling *`vTaskPrioritySet()`*.

It's also necessary to ensure that tasks yield CPU (by calling *`vTaskDelay()`*, `sleep()`, or by blocking on semaphores, queues, task notifications, etc) in order to not starve lower priority tasks and cause problems for the overall system. The *Task Watchdog Timer (TWDT)* provides a mechanism to automatically detect if task starvation happens, however note that a Task WDT timeout does not always indicate a problem (sometimes the correct operation of the firmware requires some long-running computation). In these cases tweaking the Task WDT timeout or even disabling the Task WDT may be necessary.

Built-In Task Priorities ESP-IDF starts a number of system tasks at fixed priority levels. Some are automatically started during the boot process, some are started only if the application firmware initializes a particular feature. To optimize performance, structure application task priorities so that they are not delayed by system tasks, while also not starving system tasks and impacting other functions of the system.

This may require splitting up a particular task. For example, perform a time-critical operation in a high priority task or an interrupt handler and do the non-time-critical part in a lower priority task.

Header components/esp_system/include/esp_task.h contains macros for the priority levels used for built-in ESP-IDF tasks system.

Common priorities are:

- *Main task that executes app_main function* has minimum priority (1). This task is pinned to Core 0 by default (*configurable*).
- *High Resolution Timer (ESP Timer)* system task to manage high precision timer events and execute callbacks has high priority (22, `ESP_TASK_TIMER_PRIO`). This task is pinned to Core 0.
- FreeRTOS Timer Task to handle FreeRTOS timer callbacks is created when the scheduler initializes and has minimum task priority (1, *configurable*). This task is pinned to Core 0.
- *Event Handling* system task to manage the default system event loop and execute callbacks has high priority (20, `ESP_TASK_EVENT_PRIO`) and pinned to Core 0. This configuration is only used if the application calls *`esp_event_loop_create_default()`*, it's possible to call *`esp_event_loop_create()`* with a custom task configuration instead.
- *lwIP* TCP/IP task has high priority (18, `ESP_TASK_TCPIP_PRIO`) and is not pinned to any core (*configurable*).
- *Wi-Fi Driver* task has high priority (23) and is pinned to Core 0 by default (*configurable*).
- Wi-Fi wpa_supplicant component may create dedicated tasks while the Wi-Fi Protected Setup (WPS), WPA2 EAP-TLS, Device Provisioning Protocol (DPP) or BSS Transition Management (BTM) features are in use. These tasks all have low priority (2) and are not pinned to any core.
- *Bluetooth Controller* task has high priority (23, `ESP_TASK_BT_CONTROLLER_PRIO`) and is pinned to Core 0 by default (*configurable*). The Bluetooth Controller needs to respond to requests with low latency, so it should always be close to the highest priority task assigned to a single CPU.
- *NimBLE Bluetooth Host* host task has high priority (21) and is pinned to Core 0 by default (*configurable*).
- ***Bluedroid Bluetooth Host* creates multiple tasks when used:**

  - Stack event callback task ("BTC") has high priority (19).
  - Stack BTU layer task has high priority (20).
  - Host HCI host task has high priority (22).

  All Bluedroid Tasks are pinned to the same core, which is Core 0 by default (*configurable*).
- The Ethernet driver creates a task for the MAC to receive Ethernet frames. If using the default config `ETH_MAC_DEFAULT_CONFIG` then the priority is medium-high (15) and the task is not pinned to any core. These settings can be changed by passing a custom `eth_mac_config_t` struct when initializing the Ethernet MAC.
- If using the *MQTT* component, it creates a task with default priority 5 (*configurable*, depends on *CONFIG_MQTT_USE_CUSTOM_CONFIG*) and not pinned to any core (*configurable*).
- To see what is the task priority for `mDNS` service, please check Performance Optimization.

Choosing application task priorities With a few exceptions (most importantly the lwIP TCP/IP task), in the default configuration most built-in tasks are pinned to Core 0. This makes it quite easy for the application to place high priority tasks on Core 1. Using priority 19 or higher will guarantee an application task can run on Core 1 without being preempted by any built-in task. To further isolate the tasks running on each CPU, configure the *lwIP task* to only run on Core 0 instead of either core (this may reduce total TCP/IP throughput depending on what other tasks are running).

In general, it's not recommended to set task priorities on Core 0 higher than the built-in Wi-Fi/BT operations as starving them of CPU may make the system unstable. Choosing priority 19 and Core 0 will allow lower layer Wi-Fi/BT functionality to run without delays, but still pre-empts the lwIP TCP/IP stack and other less time-critical internal functionality - this is an option for time-critical tasks that don't perform network operations. Any task that does TCP/IP network operations should run at lower priority than the lwIP TCP/IP task (18) to avoid priority inversion issues.

Note: Setting a task to always run in preference to built-in ESP-IDF tasks does not require pinning to Core 1. The task can be left unpinned - at priority 17 or lower - to optionally run on Core 0 as well, if no higher priority built-in task is running there. Using unpinned tasks can improve the overall CPU utilization, however it makes reasoning about task scheduling more complex.

Note: Task execution is always completely suspended when writing to the built-in SPI flash chip. Only *IRAM-Safe Interrupt Handlers* will continue executing.

Improving Interrupt Performance ESP-IDF supports dynamic *Interrupt allocation* with interrupt preemption. Each interrupt in the system has a priority, and higher priority interrupts will preempt lower priority ones.

Interrupt handlers will execute in preference to any task (provided the task is not inside a critical section). For this reason, it's important to minimize the amount of time spent executing in an interrupt handler.

To obtain the best performance for a particular interrupt handler:

- Assign more important interrupts a higher priority using a flag such as `ESP_INTR_FLAG_LEVEL2` or `ESP_INTR_FLAG_LEVEL3` when calling `esp_intr_alloc()`.
- Assign the interrupt on a CPU where built-in Wi-Fi/BT tasks are not configured to run (this means assigning on Core 1 by default, see *Built-In Task Priorities*). Interrupts are assigned on the same CPU where the `esp_intr_alloc()` function call is made.
- If you're sure the entire interrupt handler can run from IRAM (see *IRAM-Safe Interrupt Handlers*) then set the `ESP_INTR_FLAG_IRAM` flag when calling `esp_intr_alloc()` to assign the interrupt. This prevents it being temporarily disabled if the application firmware writes to the internal SPI flash.
- Even if the interrupt handler is not IRAM safe, if it is going to be executed frequently then consider moving the handler function to IRAM anyhow. This minimizes the chance of a flash cache miss when the interrupt code is executed (see *Targeted Optimizations*). It's possible to do this without adding the `ESP_INTR_FLAG_IRAM` flag to mark the interrupt as IRAM-safe, if only part of the handler is guaranteed to be in IRAM.

Improving Network Speed

- For Wi-Fi, see *How to Improve Wi-Fi Performance* and *Wi-Fi Buffer Usage*
- For lwIP TCP/IP (Wi-Fi and Ethernet), see *Performance Optimization*
- The wifi/iperf example contains a configuration that is heavily optimized for Wi-Fi TCP/IP throughput. Append the contents of the files wifi/iperf/sdkconfig.defaults, wifi/iperf/sdkconfig.defaults.esp32 and wifi/iperf/sdkconfig.ci.99 to your project `sdkconfig` file in order to add all of these options. Note that some of these options may have trade-offs in terms of reduced debuggability, increased firmware size, increased memory usage, or reduced performance of other features. To get the best result, read the documentation pages linked above and use this information to determine exactly which options are best suited for your app.
- The ethernet/iperf example contains a configuration that is heavily optimized for Ethernet TCP/IP throughput. Examine ethernet/iperf/sdkconfig.defaults for more details. Note that some of these options may have trade-offs in terms of reduced debuggability, increased firmware size, increased memory usage, or reduced performance of other features. To get the best result, read the documentation pages linked above and use related information to determine exactly which options are best suited for your app.

Minimizing Binary Size

The ESP-IDF build system compiles all source files in the project and ESP-IDF, but only functions and variables that are actually referenced by the program are linked into the final binary. In some cases, it is necessary to reduce the total size of the firmware binary (for example, in order to fit it into the available flash partition size).

The first step to reducing the total firmware binary size is measuring what is causing the size to increase.

Measuring Static Sizes To optimize both firmware binary size and memory usage it's necessary to measure statically allocated RAM ("data", "bss"), code ("text") and read-only data ("rodata") in your project.

Using the *idf.py* sub-commands `size`, `size-components` and `size-files` provides a summary of memory used by the project:

Size Summary (idf.py size)

```
$ idf.py size
[...]
    Total sizes:
 DRAM .data size:   14956 bytes
 DRAM .bss  size:   15808 bytes
Used static DRAM:   30764 bytes ( 149972 available, 17.0% used)
Used static IRAM:   83918 bytes (  47154 available, 64.0% used)
      Flash code:  559943 bytes
    Flash rodata:  176736 bytes
Total image size:~ 835553 bytes (.bin may be padded larger)
```

This output breaks down the size of all static memory regions in the firmware binary:

- `DRAM .data size` is statically allocated RAM that is assigned to non-zero values at startup. This uses RAM (DRAM) at runtime and also uses space in the binary file.
- `DRAM .bss size` is statically allocated RAM that is assigned zero at startup. This uses RAM (DRAM) at runtime but doesn't use any space in the binary file.
- `Used static DRAM` is the total DRAM used by .data + .bss. The `available` size is the estimated amount of DRAM which will be available as heap memory at runtime (due to metadata overhead and implementation constraints, and heap allocations done by ESP-IDF during startup, the actual free heap at startup will be lower than this).
- `Used static IRAM` is the total size of executable code *executed from IRAM*. This uses space in the binary file and also reduces IRAM and/or DRAM (depending on sizes) available as heap memory at runtime. See *Optimizing IRAM Usage*.
- `Flash code` is the total size of executable code executed from flash cache (*IROM*). This uses space in the binary file.

- `Flash rodata` is the total size of read-only data loaded from flash cache (*DROM*). This uses space in the binary file.
- `Total image size` is the estimated total binary file size, which is the total of all the used memory types except for .bss.

Component Usage Summary (idf.py size-components) The summary output provided by `idf.py size` does not give enough detail to find the main contributor to excessive binary size. To analyze in more detail, use `idf.py size-components`

```
$ idf.py size-components
[...]
    Total sizes:
 DRAM .data size:   14956 bytes
 DRAM .bss  size:   15808 bytes
Used static DRAM:   30764 bytes ( 149972 available, 17.0% used)
Used static IRAM:   83918 bytes (  47154 available, 64.0% used)
      Flash code:  559943 bytes
    Flash rodata:  176736 bytes
Total image size:~ 835553 bytes (.bin may be padded larger)
Per-archive contributions to ELF file:
            Archive File DRAM .data & .bss & other   IRAM   D/IRAM Flash code &␣
↪rodata   Total
           libnet80211.a       1267   6044       0   5490        0     107445   ␣
↪18484  138730
              liblwip.a         21   3838       0      0        0      97465   ␣
↪16116  117440
             libmbedtls.a       60    524       0      0        0      27655   ␣
↪69907   98146
          libmbedcrypto.a       64     81       0     30        0      76645   ␣
↪11661   88481
                 libpp.a      2427   1292       0  20851        0      37208    ␣
↪4708   66486
                  libc.a         4      0       0      0        0      57056    ␣
↪6455   63515
                libphy.a      1439    715       0   7798        0      33074     ␣
↪  0   43026
     libwpa_supplicant.a        12    848       0      0        0      35505    ␣
↪1446   37811
           libfreertos.a      3104    740       0  15711        0        367    ␣
↪4228   24150
           libnvs_flash.a         0     24       0      0        0      14347    ␣
↪2924   17295
           libspi_flash.a      1562    294       0   8851        0       1840    ␣
↪1913   14460
          libesp_system.a       245    206       0   3078        0       5990    ␣
↪3817   13336
             libesp-tls.a         0      4       0      0        0       5637    ␣
↪3524    9165
[... removed some lines here ...]
             libesp_rom.a         0      0       0    112        0          0     ␣
↪  0     112
                 libcxx.a         0      0       0      0        0         47     ␣
↪  0      47
                    (exe)         0      0       0      3        0          3     ␣
↪ 12      18
              libesp_pm.a         0      0       0      0        0          8     ␣
↪  0       8
             libesp_eth.a         0      0       0      0        0          0     ␣
↪  0       0
                libmesh.a         0      0       0      0        0          0     ␣
↪  0       0
```

The first lines of output from `idf.py size-components` are the same as `idf.py size`. After this a table is printed of "per-archive contributions to ELF file". This means how much each static library archive has contributed to the final binary size.

Generally, one static library archive is built per component, although some are binary libraries included by a particular component (for example, `libnet80211.a` is included by `esp_wifi` component). There are also toolchain libraries such as `libc.a` and `libgcc.a` listed here, these provide Standard C/C++ Library and toolchain built-in functionality.

If your project is simple and only has a "main" component, then all of the project's code will be shown under `libmain.a`. If your project includes its own components (see *Build System*), then they will each be shown on a separate line.

The table is sorted in descending order of the total contribution to the binary size.

The columns are as follows:

- `DRAM .data & .bss & other` - .data and .bss are the same as for the totals shown above (static variables, these both reduce total available RAM at runtime but .bss doesn't contribute to the binary file size). "other" is a column for any custom section types that also contribute to RAM size (usually this value is 0).
- `IRAM` - is the same as for the totals shown above (code linked to execute from IRAM, uses space in the binary file and also reduces IRAM that can be dynamically allocated at runtime using `HEAP_CAP_32BIT`.
- `D/IRAM` - Shows IRAM space which, due to occupying D/IRAM space, is also reducing available DRAM available as heap at runtime.
- `Flash code & rodata` - these are the same as the totals above, IROM and DROM space accessed from flash cache that contribute to the binary size.

Source File Usage Summary (idf.py size-files) For even more detail, run `idf.py size-files` to get a summary of the contribution each object file has made to the final binary size. Each object file corresponds to a single source file.

```
$ idf.py size-files
[...]
Total sizes:
 DRAM .data size:   14956 bytes
 DRAM .bss  size:   15808 bytes
Used static DRAM:   30764 bytes ( 149972 available, 17.0% used)
Used static IRAM:   83918 bytes (  47154 available, 64.0% used)
      Flash code:  559943 bytes
    Flash rodata:  176736 bytes
Total image size:~ 835553 bytes (.bin may be padded larger)
Per-file contributions to ELF file:
             Object File DRAM .data & .bss & other   IRAM   D/IRAM Flash code &␣
↪rodata   Total
     x509_crt_bundle.S.o          0      0       0      0        0          0   ␣
↪64212   64212
                wl_cnx.o          2   3183       0    221        0      13119   ␣
↪3286   19811
           phy_chip_v7.o        721    614       0   1642        0      16820   ␣
↪  0   19797
       ieee80211_ioctl.o        740     96       0    437        0      15325   ␣
↪2627   19225
                   pp.o       1142     45       0   8871        0       5030   ␣
↪537   15625
      ieee80211_output.o          2     20       0   2118        0      11617   ␣
↪914   14671
         ieee80211_sta.o          1     41       0   1498        0      10858   ␣
↪2218   14616
        lib_a-vfprintf.o          0      0       0      0        0      13829   ␣
↪752   14581
```

(continues on next page)

(continued from previous page)

```
      lib_a-svfprintf.o          0      0       0      0        0      13251     ␣
↪752   14003
            ssl_tls.c.o         60      0       0      0        0      12769     ␣
↪463   13292
            sockets.c.o          0    648       0      0        0      11096     ␣
↪1030   12774
                nd6.c.o          8    932       0      0        0      11515     ␣
↪314   12769
      phy_chip_v7_cal.o        477     53       0   3499        0       8561     ␣
↪  0   12590
                  pm.o         32    364       0   2673        0       7788     ␣
↪782   11639
       ieee80211_scan.o         18    288       0      0        0       8889     ␣
↪1921   11116
     lib_a-svfiprintf.o          0      0       0      0        0       9654     ␣
↪1206   10860
      lib_a-vfiprintf.o          0      0       0      0        0      10069     ␣
↪734   10803
         ieee80211_ht.o          0      4       0   1186        0       8628     ␣
↪898   10716
      phy_chip_v7_ana.o        241     48       0   2657        0       7677     ␣
↪  0   10623
             bignum.c.o          0      4       0      0        0       9652     ␣
↪752   10408
             tcp_in.c.o          0     52       0      0        0       8750     ␣
↪1282   10084
                  trc.o        664     88       0   1726        0       6245     ␣
↪1108    9831
              tasks.c.o          8    704       0   7594        0          0     ␣
↪1475    9781
         ecp_curves.c.o         28      0       0      0        0       7384     ␣
↪2325    9737
                ecp.c.o          0     64       0      0        0       8864     ␣
↪286    9214
     ieee80211_hostap.o          1     41       0      0        0       8578     ␣
↪585    9205
                 wdev.o        121    125       0   4499        0       3684     ␣
↪580    9009
            tcp_out.c.o          0      0       0      0        0       5686     ␣
↪2161    7847
                tcp.c.o          2     26       0      0        0       6161     ␣
↪1617    7806
      ieee80211_input.o          0      0       0      0        0       6797     ␣
↪973    7770
                wpa.c.o          0    656       0      0        0       6828     ␣
↪ 55    7539
[... additional lines removed ...]
```

After the summary of total sizes, a table of "Per-file contributions to ELF file" is printed.

The columns are the same as shown above for `idy.py size-components`, but this time the granularity is the contribution of each individual object file to the binary size.

For example, we can see that the file `x509_crt_bundle.S.o` contributed 64212 bytes to the total firmware size, all as `.rodata` in flash. Therefore we can guess that this application is using the *ESP x509 Certificate Bundle* feature and not using this feature would save at last this many bytes from the firmware size.

Some of the object files are linked from binary libraries and therefore you won't find a corresponding source file. To locate which component a source file belongs to, it's generally possible to search in the ESP-IDF source tree or look in the *Linker Map File* for the full path.

Comparing Two Binaries If making some changes that affect binary size, it's possible to use an ESP-IDF tool to break down the exact differences in size.

This operation isn't part of `idf.py`, it's necessary to run the `idf-size.py` Python tool directly.

To do so, first locate the linker map file in the build directory. It will have the name `PROJECTNAME.map`. The `idf-size.py` tool performs its analysis based on the output of the linker map file.

To compare with another binary, you will also need its corresponding `.map` file saved from the build directory.

For example, to compare two builds: one with the default *CONFIG_COMPILER_OPTIMIZATION* setting "Debug (-Og)" configuration and one with "Optimize for size (-Os)":

```
$ $IDF_PATH/tools/idf_size.py --diff build_Og/https_request.map build_Os/https_
↪request.map
<CURRENT> MAP file: build_Os/https_request.map
<REFERENCE> MAP file: build_Og/https_request.map
Difference is counted as <CURRENT> - <REFERENCE>, i.e. a positive number means␣
↪that <CURRENT> is larger.
Total sizes of <CURRENT>:
↪<REFERENCE>     Difference
 DRAM .data size:   14516 bytes                                                ␣
↪14956           -440
 DRAM .bss  size:   15792 bytes                                                ␣
↪15808            -16
Used static DRAM:   30308 bytes ( 150428 available, 16.8% used)                ␣
↪30764           -456 (   +456 available,      +0 total)
Used static IRAM:   78498 bytes (  52574 available, 59.9% used)                ␣
↪83918          -5420 (  +5420 available,      +0 total)
      Flash code:  509183 bytes                                               ␣
↪559943         -50760
    Flash rodata:  170592 bytes                                               ␣
↪176736          -6144
Total image size:~ 772789 bytes (.bin may be padded larger)                    ␣
↪835553         -62764
```

We can see from the "Difference" column that changing this one setting caused the whole binary to be over 60 KB smaller and over 5 KB more RAM is available.

It's also possible to use the "diff" mode to output a table of component-level (static library archive) differences:

```
$IDF_PATH/tools/idf_size.py --archives --diff build_Og/https_request.map build_
↪Oshttps_request.map
```

Also at the individual source file level:

```
$IDF_PATH/tools/idf_size.py --files --diff build_Og/https_request.map build_
↪Oshttps_request.map
```

Other options (like writing the output to a file) are available, pass `--help` to see the full list.

Showing Size When Linker Fails If too much static memory is used, then the linker will fail with an error such as `DRAM segment data does not fit, region `iram0_0_seg' overflowed by 44 bytes`, or similar.

In these cases, `idf.py size` will not succeed either. However it is possible to run `idf_size.py` manually in order to view the *partial static memory usage* (the memory usage will miss the variables which could not be linked, so there still appears to be some free space.)

The map file argument is `<projectname>.map` in the build directory

```
$IDF_PATH/tools/idf_size.py build/project_name.map
```

It is also possible to view the equivalent of `size-components` or `size-files` output:

```
$IDF_PATH/tools/idf_size.py --archives build/project_name.map
$IDF_PATH/tools/idf_size.py --files build/project_name.map
```

Linker Map File *This is an advanced analysis method, but it can be very useful. Feel free to skip ahead to :ref:`reducing-overall-size` and possibly come back to this later.*

The `idf.py size` analysis tools all work by parsing the GNU binutils "linker map file", which is a summary of everything the linker did when it created ("linked") the final firmware binary file

Linker map files themselves are plain text files, so it's possible to read them and find out exactly what the linker did. However, they are also very complex and long - often 100,000 or more lines!

The map file itself is broken into parts and each part has a heading. The parts are:

- `Archive member included to satisfy reference by file (symbol)`. This shows you: for each object file included in the link, what symbol (function or variable) was the linker searching for when it included that object file. If you're wondering why some object file in particular was included in the binary, this part may give a clue. This part can be used in conjunction with the `Cross Reference Table` at the end of the file. Note that not every object file shown in this list ends up included in the final binary, some end up in the `Discarded input sections` list instead.
- `Allocating common symbols` - This is a list of (some) global variables along with their sizes. Common symbols have a particular meaning in ELF binary files, but ESP-IDF doesn't make much use of them.
- `Discarded input sections` - These sections were read by the linker as part of an object file to be linked into the final binary, but then nothing else referred to them so they were discarded from the final binary. For ESP-IDF this list can be very long, as we compile each function and static variable to a unique section in order to minimize the final binary size (specifically ESP-IDF uses compiler options `-ffunction-sections -fdata-sections` and linker option `--gc-sections`). Items mentioned in this list *do not* contribute to the final binary.
- `Memory Configuration`, `Linker script and memory map` These two parts go together. Some of the output comes directly from the linker command line and the Linker Script, both provided by the *Build System*. The linker script is partially generated from the ESP-IDF project using the *Linker Script Generation* feature.
  As the output of the `Linker script and memory map` part of the map unfolds, you can see each symbol (function or static variable) linked into the final binary along with its address (as a 16 digit hex number), its length (also in hex), and the library and object file it was linked from (which can be used to determine the component and the source file).
  Following all of the output sections that take up space in the final `.bin` file, the `memory map` also includes some sections in the ELF file that are only used for debugging (ELF sections `.debug_*`, etc.). These don't contribute to the final binary size. You'll notice the address of these symbols is a very low number (starting from 0x0000000000000000 and counting up).
- `Cross Reference Table`. This table shows for each symbol (function or static variable), the list of object file(s) that referred to it. If you're wondering why a particular thing is included in the binary, this will help determine what included it.

  Note: Unfortunately, the `Cross Reference Table` doesn't only include symbols that made it into the final binary. It also includes symbols in discarded sections. Therefore, just because something is shown here doesn't mean that it was included in the final binary - this needs to be checked separately.

Note: Linker map files are generated by the GNU binutils linker "ld", not ESP-IDF. You can find additional information online about the linker map file format. This quick summary is written from the perspective of ESP-IDF build system in particular.

Reducing Overall Size The following configuration options will reduce the final binary size of almost any ESP-IDF project:

- Set *CONFIG_COMPILER_OPTIMIZATION* to "Optimize for size (-Os)". In some cases, "Optimize for performance (-O2)" will also reduce the binary size compared to the default. Note that if your code contains C or C++ Undefined Behaviour then increasing the compiler optimization level may expose bugs that otherwise don't happen.
- Reduce the compiled-in log output by lowering the app *CONFIG_LOG_DEFAULT_LEVEL*. If the *CONFIG_LOG_MAXIMUM_LEVEL* is changed from the default then this setting controls the binary size instead. Reducing compiled-in logging reduces the number of strings in the binary, and also the code size of the calls to logging functions.
- Set the *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL* to "Silent". This avoids compiling in a dedicated assertion string and source file name for each assert that may fail. It's still possible to find the failed assert in the code by looking at the memory address where the assertion failed.
- Besides the *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*, you can disable or silent the assertion for HAL component separately by setting *CONFIG_HAL_DEFAULT_ASSERTION_LEVEL*. It is to notice that ESP-IDF lowers HAL assertion level in bootloader to be silent even if *CONFIG_HAL_DEFAULT_ASSERTION_LEVEL* is set to full-assertion level. This is to reduce the bootloader size.
- Set *CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT*. This removes specific error messages for particular internal ESP-IDF error check macros. This may make it harder to debug some error conditions by reading the log output.
- If the binary needs to run on only certain revision(s) of ESP32, increasing *CONFIG_ESP32_REV_MIN* to match can result in a reduced binary size. This will make a large difference if setting ESP32 minimum revision 3, and PSRAM is enabled.
- Don't enable *CONFIG_COMPILER_CXX_EXCEPTIONS*, *CONFIG_COMPILER_CXX_RTTI*, or set the *CONFIG_COMPILER_STACK_CHECK_MODE* to Overall. All of these options are already disabled by default, but they have a large impact on binary size.
- Disabling *CONFIG_ESP_ERR_TO_NAME_LOOKUP* will remove the lookup table to translate user-friendly names for error values (see *Error Handling*) in error logs, etc. This saves some binary size, but error values will be printed as integers only.
- Setting *CONFIG_ESP_SYSTEM_PANIC* to "Silent reboot" will save a small amount of binary size, however this is *only* recommended if no one will use UART output to debug the device.
- If the application binary uses only one of the security versions of the protocomm component, then the support for others can be disabled to save some code size. The support can be disabled through *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0*, *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1* or *CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2* respectively.

Note: In addition to the many configuration items shown here, there are a number of configuration options where changing the option from the default will increase binary size. These are not noted here. Where the increase is significant, this is usually noted in the configuration item help text.

Targeted Optimizations The following binary size optimizations apply to a particular component or a function:

Wi-Fi

- Disabling *CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE* will save some Wi-Fi binary size if WPA3 support is not needed. (Note that WPA3 is mandatory for new Wi-Fi device certifications.)
- Disabling *CONFIG_ESP_WIFI_SOFTAP_SUPPORT* will save some Wi-Fi binary size if soft-AP support is not needed.
- Disabling *CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT* will save some Wi-Fi binary size if enterprise support is not needed.

ADC

- Disabling ADC calibration features *CONFIG_ADC_CAL_EFUSE_TP_ENABLE*, *CONFIG_ADC_CAL_EFUSE_VREF_ENABLE*, *CONFIG_ADC_CAL_LUT_ENABLE* will save a small amount of binary size if ADC driver is used, at expense of accuracy.

Bluetooth NimBLE If using *NimBLE Bluetooth Host* then the following modifications can reduce binary size:

- Set *CONFIG_BTDM_CTRL_BLE_MAX_CONN* to 1 if only one BLE connection is needed.
- *CONFIG_BT_NIMBLE_MAX_CONNECTIONS* to 1 if only one BLE connection is needed.
- Disable either *CONFIG_BT_NIMBLE_ROLE_CENTRAL* or *CONFIG_BT_NIMBLE_ROLE_OBSERVER* if these roles are not needed.
- Reducing *CONFIG_BT_NIMBLE_LOG_LEVEL* can reduce binary size. Note that if the overall log level has been reduced as described above in *Reducing Overall Size* then this also reduces the NimBLE log level.

lwIP IPv6

- Setting *CONFIG_LWIP_IPV6* to false will reduce the size of the lwIP TCP/IP stack, at the cost of only supporting IPv4.

Note: IPv6 is required by some components such as `coap` and *ASIO port*, These components will not be available if IPV6 is disabled.

Newlib nano formatting By default, ESP-IDF uses newlib "full" formating for I/O (printf, scanf, etc.)

Enabling the config option *CONFIG_NEWLIB_NANO_FORMAT* will switch newlib to the "nano" formatting mode. This both smaller in code size and a large part of the implementation is compiled into the ESP32 ROM, so it doesn't need to be included in the binary at all.

The exact difference in binary size depends on which features the firmware uses, but 25 KB ~ 50 KB is typical.

Enabling Nano formatting also reduces the stack usage of each function that calls printf() or another string formatting function, see *Reducing Stack Sizes*.

"Nano" formatting doesn't support 64-bit integers, or C99 formatting features. For a full list of restrictions, search for `--enable-newlib-nano-formatted-io` in the Newlib README file.

mbedTLS features Under *Component Config -> mbedTLS* there are multiple mbedTLS features which are enabled by default but can be disabled if not needed to save code size.

These include:

- *CONFIG_MBEDTLS_HAVE_TIME*
- *CONFIG_MBEDTLS_ECDSA_DETERMINISTIC*
- *CONFIG_MBEDTLS_SHA512_C*
- *CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_SSL_CONTEXT_SERIALIZATION*
- *CONFIG_MBEDTLS_SSL_ALPN*
- *CONFIG_MBEDTLS_SSL_RENEGOTIATION*
- *CONFIG_MBEDTLS_CCM_C*
- *CONFIG_MBEDTLS_GCM_C*
- *CONFIG_MBEDTLS_ECP_C* (Alternatively: Leave this option enabled but disable some of the elliptic curves listed in the sub-menu.)
- *CONFIG_MBEDTLS_ECP_NIST_OPTIM*
- *CONFIG_MBEDTLS_ECP_FIXED_POINT_OPTIM*
- Change *CONFIG_MBEDTLS_TLS_MODE* if both server & client functionalities are not needed
- Consider disabling some ciphersuites listed in the "TLS Key Exchange Methods" sub-menu (i.e. *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA*)

The help text for each option has some more information.

Important: It is **strongly not recommended to disable all these mbedTLS options**. Only disable options where you understand the functionality and are certain that it is not needed in the application. In particular:

- Ensure that any TLS server(s) the device connects to can still be used. If the server is controlled by a third party or a cloud service, recommend ensuring that the firmware supports at least two of the supported cipher suites in case one is disabled in a future update.
- Ensure that any TLS client(s) that connect to the device can still connect with supported/recommended cipher suites. Note that future versions of client operating systems may remove support for some features, so it is recommended to enable multiple supported cipher suites or algorithms for redundancy.

If depending on third party clients or servers, always pay attention to announcements about future changes to supported TLS features. If not, the ESP32 device may become inaccessible if support changes.

Note: Not every combination of mbedTLS compile-time config is tested in ESP-IDF. If you find a combination that fails to compile or function as expected, please report the details on GitHub.

VFS *Virtual filesystem* feature in ESP-IDF allows multiple filesystem drivers and file-like peripheral drivers to be accessed using standard I/O functions (`open`, `read`, `write`, etc.) and C library functions (`fopen`, `fread`, `fwrite`, etc.). When filesystem or file-like peripheral driver functionality is not used in the application this feature can be fully or partially disabled. VFS component provides the following configuration options:

- *CONFIG_VFS_SUPPORT_TERMIOS* —can be disabled if the application doesn't use `termios` family of functions. Currently, these functions are implemented only for UART VFS driver. Most applications can disable this option. Disabling this option reduces the code size by about 1.8 kB.
- *CONFIG_VFS_SUPPORT_SELECT* — can be disabled if the application doesn't use `select` function with file descriptors. Currently, only the UART and eventfd VFS drivers implement `select` support. Note that when this option is disabled, `select` can still be used for socket file descriptors. Disabling this option reduces the code size by about 2.7 kB.
- *CONFIG_VFS_SUPPORT_DIR* —can be disabled if the application doesn't use directory related functions, such as `readdir` (see the description of this option for the complete list). Applications which only open, read and write specific files and don't need to enumerate or create directories can disable this option, reducing the code size by 0.5 kB or more, depending on the filesystem drivers in use.
- *CONFIG_VFS_SUPPORT_IO* —can be disabled if the application doesn't use filesystems or file-like peripheral drivers. This disables all VFS functionality, including the three options mentioned above. When this option is disabled, *console* can't be used. Note that the application can still use standard I/O functions with socket file descriptors when this option is disabled. Compared to the default configuration, disabling this option reduces code size by about 9.4 kB.

Bootloader Size This document deals with the size of an ESP-IDF app binary only, and not the ESP-IDF *Second stage bootloader*.

For a discussion of ESP-IDF bootloader binary size, see *Bootloader Size*.

IRAM Binary Size If the IRAM section of a binary is too large, this issue can be resolved by reducing IRAM memory usage. See *Optimizing IRAM Usage*.

Minimizing RAM Usage

In some cases, a firmware application's available RAM may run low or run out entirely. In these cases, it's necessary to tune the memory usage of the firmware application.

In general, firmware should aim to leave some "headroom" of free internal RAM in order to deal with extraordinary situations or changes in RAM usage in future updates.

Background Before optimizing ESP-IDF RAM usage, it's necessary to understand the basics of ESP32 memory types, the difference between static and dynamic memory usage in C, and the way ESP-IDF uses stack and heap. This information can all be found in *Heap Memory Allocation*.

Measuring Static Memory Usage The *idf.py* tool can be used to generate reports about the static memory usage of an application. Refer to *the Binary Size chapter for more information*.

Measuring Dynamic Memory Usage ESP-IDF contains a range of heap APIs for measuring free heap at runtime. See *Heap Memory Debugging*.

Note: In embedded systems, heap fragmentation can be a significant issue alongside total RAM usage. The heap measurement APIs provide ways to measure the "largest free block". Monitoring this value along with the total number of free bytes can give a quick indication of whether heap fragmentation is becoming an issue.

Reducing Static Memory Usage

- Reducing the static memory usage of the application increases the amount of RAM available for heap at runtime, and vice versa.
- Generally speaking, minimizing static memory usage requires monitoring the .data and .bss sizes. For tools to do this, see *Measuring Static Sizes*.
- Internal ESP-IDF functions do not make heavy use of static RAM allocation in C. In many instances (including: Wi-Fi library, Bluetooth controller) "static" buffers are still allocated from heap, but the allocation is done once when the feature is initialized and will be freed if the feature is deinitialized. This is done in order to maximize the amount of free memory at different points in the application life-cycle.

To minimize static memory use:

- Declare structures, buffers, or other variables `const` whenever possible. Constant data can be stored in flash not RAM. This may require changing functions in the firmware to take `const *` arguments instead of mutable pointer arguments. These changes can also reduce the stack usage of some functions.
- If using Bluedroid, setting the option *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY* will cause Bluedroid to allocate memory on initialization and free it on deinitialization. This doesn't necessarily reduce the peak memory usage, but changes it from static memory usage to runtime memory usage.
- If *Coredump* component is enabled, *ESP_COREDUMP_LOG* macros will use ~5KB internal memory to place strings into DRAM. By disabling *CONFIG_ESP_COREDUMP_LOGS* option, these logs are disabled and the memory is reclaimed.

Reducing Stack Sizes In FreeRTOS, task stacks are usually allocated from the heap. The stack size for each task is fixed (passed as an argument to `xTaskCreate()`). Each task can use up to its allocated stack size, but using more than this will cause an otherwise valid program to crash with a stack overflow or heap corruption.

Therefore, determining the optimum sizes of each task stack can substantially reduce RAM usage.

To determine optimum task stack sizes:

- Combine tasks. The best task stack size is 0 bytes, achieved by combining a task with another existing task. Anywhere that the firmware can be structured to perform multiple functions sequentially in a single task will increase free memory. In some cases, using a "worker task" pattern where jobs are serialized into a FreeRTOS queue (or similar) and then processed by generic worker tasks may help.
- Consolidate task functions. String formatting functions (like `printf`) are particularly heavy users of stack, so any task which doesn't ever call these can usually have its stack size reduced.
- Enabling *Newlib nano formatting* will reduce the stack usage of any task that calls `printf()` or other C string formatting functions.
- Avoid allocating large variables on the stack. In C, any large struct or array allocated as an "automatic" variable (i.e. default scope of a C declaration) will use space on the stack. Minimize the sizes of these, allocate them statically and/or see if you can save memory by allocating them from the heap only when they are needed.
- Avoid deep recursive function calls. Individual recursive function calls don't always add a lot of stack usage each time they are called, but if each function includes large stack-based variables then the overhead can get quite high.

- At runtime, call the function *uxTaskGetStackHighWaterMark()* with the handle of any task where you think there is unused stack memory. This function returns the minimum lifetime free stack memory in bytes. The easiest time to call this is from the task itself: call `uxTaskGetStackHighWaterMark(NULL)` to get the current task's high water mark after the time that the task has achieved its peak stack usage (i.e. if there is a main loop, execute the main loop a number of times with all possible states and then call *uxTaskGetStackHighWaterMark()*). Often, it's possible to subtract almost the entire value returned here from the total stack size of a task, but allow some safety margin to account for unexpected small increases in stack usage at runtime.
- Call *uxTaskGetSystemState()* at runtime to get a summary of all tasks in the system. This includes their individual stack "high watermark" values.
- When debugger watchpoints are not being used, set the *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK* option to trigger an immediate panic if a task writes the word at the end of its assigned stack. This is slightly more reliable than the default *CONFIG_FREERTOS_CHECK_STACKOVERFLOW* option of "Check using canary bytes", because the panic happens immediately, not on the next RTOS context switch. Neither option is perfect, it's possible in some cases for stack pointer to skip the watchpoint or canary bytes and corrupt another region of RAM, instead.

Internal Stack Sizes ESP-IDF allocates a number of internal tasks for housekeeping purposes or operating system functions. Some are created during the startup process, and some are created at runtime when particular features are initialized.

The default stack sizes for these tasks are usually set conservatively high, to allow all common usage patterns. Many of the stack sizes are configurable, and it may be possible to reduce them to match the real runtime stack usage of the task.

Important: If internal task stack sizes are set too small, ESP-IDF will crash unpredictably. Even if the root cause is task stack overflow, this is not always clear when debugging. It is recommended that internal stack sizes are only reduced carefully (if at all), with close attention to "high water mark" free space under load. If reporting an issue that occurs when internal task stack sizes have been reduced, please always include this information and the specific configuration that is being used.

- *Main task that executes app_main function* has stack size *CONFIG_ESP_MAIN_TASK_STACK_SIZE*.
- *High Resolution Timer (ESP Timer)* system task which executes callbacks has stack size *CONFIG_ESP_TIMER_TASK_STACK_SIZE*.
- FreeRTOS Timer Task to handle FreeRTOS timer callbacks has stack size *CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH*.
- *Event Handling* system task to execute callbacks for the default system event loop has stack size *CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE*.
- *lwIP* TCP/IP task has stack size *CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*
- *Bluedroid Bluetooth Host* have task stack sizes *CONFIG_BT_BTC_TASK_STACK_SIZE*, *CONFIG_BT_BTU_TASK_STACK_SIZE*.
- *NimBLE Bluetooth Host* has task stack size *CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE*
- The Ethernet driver creates a task for the MAC to receive Ethernet frames. If using the default config `ETH_MAC_DEFAULT_CONFIG` then the task stack size is 4 KB. This setting can be changed by passing a custom *eth_mac_config_t* struct when initializing the Ethernet MAC.
- FreeRTOS idle task stack size is configured by *CONFIG_FREERTOS_IDLE_TASK_STACKSIZE*.
- If using the *MQTT* component, it creates a task with stack size configured by *CONFIG_MQTT_TASK_STACK_SIZE*. MQTT stack size can also be configured using `task_stack` field of *esp_mqtt_client_config_t*.
- To see how to optimize RAM usage when using `mDNS`, please check Performance Optimization.

Note: Aside from built-in system features such as esp-timer, if an ESP-IDF feature is not initialized by the firmware then no associated task is created. In those cases, the stack usage is zero and the stack size configuration for the task is not relevant.

Reducing Heap Usage For functions that assist in analyzing heap usage at runtime, see *Heap Memory Debugging*.

Normally, optimizing heap usage consists of analyzing the usage and removing calls to `malloc()` that aren't being used, reducing the corresponding sizes, or freeing previously allocated buffers earlier.

There are some ESP-IDF configuration options that can reduce heap usage at runtime:

- lwIP documentation has a section to configure *Minimum RAM usage*.
- *Wi-Fi Buffer Usage* describes options to either reduce numbers of "static" buffers or reduce the maximum number of "dynamic" buffers in use, in order to minimize memory usage at possible cost of performance. Note that "static" Wi-Fi buffers are still allocated from heap when Wi-Fi is initialized and will be freed if Wi-Fi is deinitialized.
- The Ethernet driver allocates DMA buffers for the internal Ethernet MAC when it is initialized - configuration options are *CONFIG_ETH_DMA_BUFFER_SIZE*, *CONFIG_ETH_DMA_RX_BUFFER_NUM*, *CONFIG_ETH_DMA_TX_BUFFER_NUM*.
- Several Mbed TLS configuration options can be used to reduce heap memory usage. See the *Mbed TLS* docs for details.
- In single core mode only, it's possible to use IRAM as byte accessible memory (added to the regular heap) by enabling *CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY*. Note that this option carries a performance penalty and the risk of security issues caused by executable data. If this option is enabled then it's possible to set other options to prefer certain buffers be allocated from this memory: *mbedTLS*, *NimBLE*.
- Reduce *CONFIG_BTDM_CTRL_BLE_MAX_CONN* if using BLE.
- Reduce *CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN* if using Bluetooth Classic.

Note: There are other configuration options that will increase heap usage at runtime if changed from the defaults. These are not listed here, but the help text for the configuration item will mention if there is some memory impact.

Optimizing IRAM Usage If the app allocates more static IRAM than is available then the app will fail to build and linker errors such as `section `.iram0.text' will not fit in region `iram0_0_seg'`, `IRAM0 segment data does not fit` and `region `iram0_0_seg' overflowed by 84 bytes` will be seen. If this happens, it is necessary to find ways to reduce static IRAM usage in order to link the application.

To analyze the IRAM usage in the firmware binary, use *Measuring Static Sizes*. If the firmware failed to link, steps to analyze are shown at *Showing Size When Linker Fails*.

The following options will reduce IRAM usage of some ESP-IDF features:

- Enable *CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH*. Provided these functions are not (incorrectly) used from ISRs, this option is safe to enable in all configurations.
- Enable *CONFIG_FREERTOS_PLACE_SNAPSHOT_FUNS_INTO_FLASH*. Enabling this option will place snapshot-related functions, such as `vTaskGetSnapshot` or `uxTaskGetSnapshotAll`, in flash.
- Enable *CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH*. Provided these functions are not (incorrectly) used from ISRs, this option is safe to enable in all configurations.
- Enable *CONFIG_RINGBUF_PLACE_ISR_FUNCTIONS_INTO_FLASH*. This option is not safe to use if the ISR ringbuf functions are used from an IRAM interrupt context, e.g. if *CONFIG_UART_ISR_IN_IRAM* is enabled. For the IDF drivers where this is the case you will get an error at run-time when installing the driver in question.
- Disable Wi-Fi options *CONFIG_ESP32_WIFI_IRAM_OPT* and/or *CONFIG_ESP32_WIFI_RX_IRAM_OPT*. Disabling these options will free available IRAM at the cost of Wi-Fi performance.
- *CONFIG_SPI_FLASH_ROM_DRIVER_PATCH* disabling this option will free some IRAM but is only available in some flash configurations (see the configuration item help text).
- If the application uses PSRAM and is based on ESP32 rev. 3 (ECO3), setting *CONFIG_ESP32_REV_MIN* to `3` will disable PSRAM bug workarounds, saving ~10kB or more of IRAM.
- Disabling *CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR* prevents posting `esp_event` events from *IRAM-Safe Interrupt Handlers* but will save some IRAM.
- Disabling *CONFIG_SPI_MASTER_ISR_IN_IRAM* prevents spi_master interrupts from being serviced while writing to flash, and may otherwise reduce spi_master performance, but will save some IRAM.

- Setting *CONFIG_HAL_DEFAULT_ASSERTION_LEVEL* to disable assertion for HAL component will save some IRAM especially for HAL code who calls *HAL_ASSERT* a lot and resides in IRAM.

Putting C Library in Flash When compiling for ESP32 revisions older than ECO3 (*CONFIG_ESP32_REV_MIN*), PSRAM cache bug workaround (*CONFIG_SPIRAM_CACHE_WORKAROUND*) option is enabled, and the C library functions normally located in ROM are recompiled with the workaround and placed into IRAM instead. For most applications, it is safe to move many of the C library functions into Flash, reclaiming some IRAM. Corresponding options include:

- *CONFIG_SPIRAM_CACHE_LIBJMP_IN_IRAM*: affects the functions `longjmp` and `setjump`.
- *CONFIG_SPIRAM_CACHE_LIBMATH_IN_IRAM*: affects the functions `abs`, `div`, `labs`, `ldiv`, `quorem`, `fpclassify` and `nan`.
- *CONFIG_SPIRAM_CACHE_LIBNUMPARSER_IN_IRAM*: affects the functions `utoa`, `itoa`, `atoi`, `atol`, `strtol`, and `strtoul`.
- *CONFIG_SPIRAM_CACHE_LIBIO_IN_IRAM*: affects the functions `wcrtomb`, `fvwrite`, `wbuf`, `wsetup`, `fputwc`, `wctomb_r`, `ungetc`, `makebuf`, `fflush`, `refill`, and `sccl`.
- *CONFIG_SPIRAM_CACHE_LIBTIME_IN_IRAM*: affects the functions `asctime`, `asctime_r`, `ctime`, `ctime_r`, `lcltime`, `lcltime_r`, `gmtime`, `gmtime_r`, `strftime`, `mktime`, `tzset_r`, `tzset`, `time`, `gettzinfo`, `systimes`, `month_lengths`, `timelocal`, `tzvars`, `tzlock`, `tzcalc_limits`, and `strptime`.
- *CONFIG_SPIRAM_CACHE_LIBCHAR_IN_IRAM*: affects the functions `ctype_`, `toupper`, `tolower`, `toascii`, `strupr`, `bzero`, `isalnum`, `isalpha`, `isascii`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, and `isupper`.
- *CONFIG_SPIRAM_CACHE_LIBMEM_IN_IRAM*: affects the functions `memccpy`, `memchr`, `memmove`, and `memrchr`.
- *CONFIG_SPIRAM_CACHE_LIBSTR_IN_IRAM*: affects the functions `strcasecmp`, `strcasestr`, `strchr`, `strcoll`, `strcpy`, `strcspn`, `strdup`, `strdup_r`, `strlcat`, `strlcpy`, `strlen`, `strlwr`, `strncasecmp`, `strncat`, `strncmp`, `strncpy`, `strndup`, `strndup_r`, `strrchr`, `strsep`, `strspn`, `strstr`, `strtok_r`, and ``strupr`.
- *CONFIG_SPIRAM_CACHE_LIBRAND_IN_IRAM*: affects the functions `srand`, `rand`, and `rand_r`.
- *CONFIG_SPIRAM_CACHE_LIBENV_IN_IRAM*: affects the functions `environ`, `envlock`, and `getenv_r`.
- *CONFIG_SPIRAM_CACHE_LIBFILE_IN_IRAM*: affects the functions lock``, `isatty`, `fclose`, `open`, `close`, `creat`, `read`, `rshift`, `sbrk`, `stdio`, `syssbrk`, `sysclose`, `sysopen`, `creat`, `sysread`, `syswrite`, `impure`, `fwalk`, and `findfp`.
- *CONFIG_SPIRAM_CACHE_LIBMISC_IN_IRAM*: affects the functions `raise` and `system`.

The exact amount of IRAM saved will depend on how much C library code is actually used by the application. In addition to these, the following options may be used to move more of the C library code into Flash, however note that this may result in reduced performance. Also take care to not use corresponding C library functions from interrupts which may be called while cache is disabled (allocated with *ESP_INTR_FLAG_IRAM* flag), refer to *IRAM-Safe Interrupt Handlers* for more details. For these reasons, the functions `itoa`, `memcmp`, `memcpy`, `memset`, `strcat`, `strcmp`, and `strlen` are always put in IRAM.

Note: Moving frequently-called functions from IRAM to flash may increase their execution time.

Note: Other configuration options exist that will increase IRAM usage by moving some functionality into IRAM, usually for performance, but the default option is not to do this. These are not listed here. The IRAM size impact of enabling these options is usually noted in the configuration item help text.

4.23 RF calibration

ESP32 supports three RF calibration methods during RF initialization:

1. Partial calibration
2. Full calibration
3. No calibration

4.23.1 Partial calibration

During RF initialization, the partial calibration method is used by default for RF calibration. It is done based on the full calibration data which is stored in the NVS. To use this method, please go to `menuconfig` and enable *CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE*.

4.23.2 Full calibration

Full calibration is triggered in the follwing conditions:

1. NVS does not exist.
2. The NVS partition to store calibration data is erased.
3. Hardware MAC address is changed.
4. PHY library version is changed.
5. The RF calibration data loaded from the NVS partition is broken.

It takes about 100ms more than partial calibration. If boot duration is not critical, it is suggested to use the full calibration method. To switch to the full calibration method, go to `menuconfig` and disable *CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE*. If you use the default method of RF calibration, there are two ways to add the function of triggering full calibration as a last-resort remedy.

1. Erase the NVS partition if you don't mind all of the data stored in the NVS partition is erased. That is indeed the easiest way.
2. Call API `esp_phy_erase_cal_data_in_nvs()` before initializing WiFi and BT/BLE based on some conditions (e.g. an option provided in some diagnostic mode). In this case, only phy namespace of the NVS partition is erased.

4.23.3 No calibration

No calibration method is only used when the device wakes up from deep sleep.

4.23.4 PHY initialization data

The PHY initialization data is used for RF calibration. There are two ways to get the PHY initialization data.

One is the default initialization data which is located in the header file components/esp_phy/esp32/include/phy_init_data.h.

It is embedded into the application binary after compiling and then stored into read-only memory (DROM). To use the default initialization data, please go to `menuconfig` and disable *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*.

Another is the initialization data which is stored in a partition. When using a custom partition table, make sure that PHY data partition is included (type: *data*, subtype: *phy*). With default partition table, this is done automatically. If initialization data is stored in a partition, it has to be flashed there, otherwise runtime error will occur. To switch to the initialization data stored in a partition, go to `menuconfig` and enable *CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*.

4.23.5 API Reference

Header File

- components/esp_phy/include/esp_phy_init.h

Functions

const *esp_phy_init_data_t* ***esp_phy_get_init_data** (void)

Get PHY init data.

If "Use a partition to store PHY init data" option is set in menuconfig, This function will load PHY init data from a partition. Otherwise, PHY init data will be compiled into the application itself, and this function will return a pointer to PHY init data located in read-only memory (DROM).

If "Use a partition to store PHY init data" option is enabled, this function may return NULL if the data loaded from flash is not valid.

Note: Call esp_phy_release_init_data to release the pointer obtained using this function after the call to esp_wifi_init.

Returns pointer to PHY init data structure

void **esp_phy_release_init_data** (const *esp_phy_init_data_t* *data)

Release PHY init data.

Parameters **data** – pointer to PHY init data structure obtained from esp_phy_get_init_data function

esp_err_t **esp_phy_load_cal_data_from_nvs** (*esp_phy_calibration_data_t* *out_cal_data)

Function called by esp_phy_load_cal_and_init to load PHY calibration data.

This is a convenience function which can be used to load PHY calibration data from NVS. Data can be stored to NVS using esp_phy_store_cal_data_to_nvs function.

If calibration data is not present in the NVS, or data is not valid (was obtained for a chip with a different MAC address, or obtained for a different version of software), this function will return an error.

Parameters **out_cal_data** – pointer to calibration data structure to be filled with loaded data.
Returns ESP_OK on success

esp_err_t **esp_phy_store_cal_data_to_nvs** (const *esp_phy_calibration_data_t* *cal_data)

Function called by esp_phy_load_cal_and_init to store PHY calibration data.

This is a convenience function which can be used to store PHY calibration data to the NVS. Calibration data is returned by esp_phy_load_cal_and_init function. Data saved using this function to the NVS can later be loaded using esp_phy_store_cal_data_to_nvs function.

Parameters **cal_data** – pointer to calibration data which has to be saved.
Returns ESP_OK on success

esp_err_t **esp_phy_erase_cal_data_in_nvs** (void)

Erase PHY calibration data which is stored in the NVS.

This is a function which can be used to trigger full calibration as a last-resort remedy if partial calibration is used. It can be called in the application based on some conditions (e.g. an option provided in some diagnostic mode).

Returns ESP_OK on success
Returns others on fail. Please refer to NVS API return value error number.

bool **esp_phy_is_initialized**(void)

Get phy initialize status.

Returns return true if phy is already initialized.

void **esp_phy_enable**(void)

Enable PHY and RF module.

PHY and RF module should be enabled in order to use WiFi or BT. Now PHY and RF enabling job is done automatically when start WiFi or BT. Users should not call this API in their application.

void **esp_phy_disable**(void)

Disable PHY and RF module.

PHY module should be disabled in order to shutdown WiFi or BT. Now PHY and RF disabling job is done automatically when stop WiFi or BT. Users should not call this API in their application.

void **esp_phy_load_cal_and_init**(void)

Load calibration data from NVS and initialize PHY and RF module.

void **esp_phy_modem_init**(void)

Initialize backup memory for Phy power up/down.

void **esp_phy_modem_deinit**(void)

Deinitialize backup memory for Phy power up/down Set phy_init_flag if all modems deinit on ESP32C3.

void **esp_phy_common_clock_enable**(void)

Enable WiFi/BT common clock.

void **esp_phy_common_clock_disable**(void)

Disable WiFi/BT common clock.

int64_t **esp_phy_rf_get_on_ts**(void)

Get the time stamp when PHY/RF was switched on.

Returns return 0 if PHY/RF is never switched on. Otherwise return time in microsecond since boot when phy/rf was last switched on

esp_err_t **esp_phy_update_country_info**(const char *country)

Update the corresponding PHY init type according to the country code of Wi-Fi.

Parameters **country** –country code
Returns ESP_OK on success.
Returns esp_err_t code describing the error on fail

char ***get_phy_version_str**(void)

Get PHY lib version.

Returns PHY lib version.

void **phy_init_param_set**(uint8_t param)

Set PHY init parameters.

Parameters **param** –is 1 means combo module

void **phy_wifi_enable_set**(uint8_t enable)

Wi-Fi RX enable.

Parameters **enable** –Whether to enable phy for wifi

Structures

struct **esp_phy_init_data_t**

Structure holding PHY init parameters.

Public Members

uint8_t **params**[128]

opaque PHY initialization parameters

struct **esp_phy_calibration_data_t**

Opaque PHY calibration data.

Public Members

uint8_t **version**[4]

PHY version

uint8_t **mac**[6]

The MAC address of the station

uint8_t **opaque**[1894]

calibration data

Enumerations

enum **esp_phy_calibration_mode_t**

PHY calibration mode.

Values:

enumerator **PHY_RF_CAL_PARTIAL**

Do part of RF calibration. This should be used after power-on reset.

enumerator **PHY_RF_CAL_NONE**

Don't do any RF calibration. This mode is only suggested to be used after deep sleep reset.

enumerator **PHY_RF_CAL_FULL**

Do full RF calibration. Produces best results, but also consumes a lot of time and current. Suggested to be used once.

4.24 Secure Boot

Important: All references in this document are related to Secure Boot V1 (The AES based Secure Boot Scheme). ESP32 Revision 3 onwards, the preferred secure boot scheme is *Secure Boot V2*. Please refer to Secure Boot V2 document for ESP32 Revision 3 or ESP32-S2.

Secure Boot is a feature for ensuring only your code can run on the chip. Data loaded from flash is verified on each reset.

Secure Boot is separate from the *Flash Encryption* feature, and you can use secure boot without encrypting the flash contents. However, for a secure environment both should be used simultaneously. See *Secure Boot & Flash Encryption* for more details.

Important: Enabling secure boot limits your options for further updates of your ESP32. Make sure to read this document throughly and understand the implications of enabling secure boot.

4.24.1 Background

- Most data is stored in flash. Flash access does not need to be protected from physical access in order for secure boot to function, because critical data is stored (non-software-accessible) in Efuses internal to the chip.
- Efuses are used to store the secure bootloader key (in efuse BLOCK2), and also a single Efuse bit (ABS_DONE_0) is burned (written to 1) to permanently enable secure boot on the chip. For more details on eFuses, see *ESP32 Technical Reference Manual > eFuse Controller (eFuse)* [PDF].
- To understand the secure boot process, first familiarise yourself with the standard *ESP-IDF boot process*.
- Both stages of the boot process (initial software bootloader load, and subsequent partition & app loading) are verified by the secure boot process, in a "chain of trust" relationship.

4.24.2 Secure Boot Process Overview

This is a high level overview of the secure boot process. Step by step instructions are supplied under *How To Enable Secure Boot*. Further in-depth details are supplied under *Technical Details*:

1. The options to enable secure boot are provided in the *Project Configuration Menu*, under "Secure Boot Configuration".
2. Secure Boot defaults to signing images and partition table data during the build process. The "Secure boot private signing key" config item is a file path to a ECDSA public/private key pair in a PEM format file.
3. The software bootloader image is built by esp-idf with secure boot support enabled and the public key (signature verification) portion of the secure boot signing key compiled in. This software bootloader image is flashed at offset 0x1000.
4. On first boot, the software bootloader follows the following process to enable secure boot:
   - Hardware secure boot support generates a device secure bootloader key (generated via hardware RNG, then stored read/write protected in efuse), and a secure digest. The digest is derived from the key, an IV, and the bootloader image contents.
   - The secure digest is flashed at offset 0x0 in the flash.
   - Depending on Secure Boot Configuration, efuses are burned to disable JTAG and the ROM BASIC interpreter (it is strongly recommended these options are turned on.)
   - Bootloader permanently enables secure boot by burning the ABS_DONE_0 efuse. The software bootloader then becomes protected (the chip will only boot a bootloader image if the digest matches.)
5. On subsequent boots the ROM bootloader sees that the secure boot efuse is burned, reads the saved digest at 0x0 and uses hardware secure boot support to compare it with a newly calculated digest. If the digest does not match then booting will not continue. The digest and comparison are performed entirely by hardware, and the calculated digest is not readable by software. For technical details see *Secure Boot Hardware Support*.
6. When running in secure boot mode, the software bootloader uses the secure boot signing key (the public key of which is embedded in the bootloader itself, and therefore validated as part of the bootloader) to verify the signature appended to all subsequent partition tables and app images before they are booted.

4.24.3 Keys

The following keys are used by the secure boot process:

- "secure bootloader key" is a 256-bit AES key that is stored in Efuse block 2. The bootloader can generate this key itself from the internal hardware random number generator, the user does not need to supply it (it is optionally possible to supply this key, see *Re-Flashable Software Bootloader*). The Efuse holding this key is read & write protected (preventing software access) before secure boot is enabled.
  - By default, the Efuse Block 2 Coding Scheme is "None" and a 256 bit key is stored in this block. On some ESP32s, the Coding Scheme is set to 3/4 Encoding (CODING_SCHEME efuse has value 1) and a 192 bit key must be stored in this block.

For more details, see *ESP32 Technical Reference Manual* > *eFuse Controller (eFuse)* > *System Parameter coding_scheme* [PDF].
The algorithm operates on a 256 bit key in all cases, 192 bit keys are extended by repeating some bits (*details*).

- "secure boot signing key" is a standard ECDSA public/private key pair (see *Image Signing Algorithm*) in PEM format.
  - The public key from this key pair (for signature verification but not signature creation) is compiled into the software bootloader and used to verify the second stage of booting (partition table, app image) before booting continues. The public key can be freely distributed, it does not need to be kept secret.
  - The private key from this key pair *must be securely kept private*, as anyone who has this key can authenticate to any bootloader that is configured with secure boot and the matching public key.

4.24.4 Bootloader Size

Enabling Secure boot and/or flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

4.24.5 How To Enable Secure Boot

1. Open the *Project Configuration Menu*, navigate to "Secure Boot Configuration" and select the option "One-time Flash". (To understand the alternative "Reflashable" choice, see *Re-Flashable Software Bootloader*.)
2. Select a name for the secure boot signing key. This option will appear after secure boot is enabled. The file can be anywhere on your system. A relative path will be evaluated from the project directory. The file does not need to exist yet.
3. Set other menuconfig options (as desired). Pay particular attention to the "Bootloader Config" options, as you can only flash the bootloader once. Then exit menuconfig and save your configuration
4. The first time you run `make`, if the signing key is not found then an error message will be printed with a command to generate a signing key via `espsecure.py generate_signing_key`.

Important: A signing key generated this way will use the best random number source available to the OS and its Python installation (*/dev/urandom* on OSX/Linux and *CryptGenRandom()* on Windows). If this random number source is weak, then the private key will be weak.

Important: For production environments, we recommend generating the keypair using openssl or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.

---

5. Run `idf.py bootloader` to build a secure boot enabled bootloader. The build output will include a prompt for a flashing command, using `esptool.py write_flash`.
6. When you're ready to flash the bootloader, run the specified command (you have to enter it yourself, this step is not performed by make) and then wait for flashing to complete. **Remember this is a one time flash, you can't change the bootloader after this!**.
7. Run `idf.py flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 4.

Note: `idf.py flash` doesn't flash the bootloader if secure boot is enabled.

---

8. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable secure boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that secure boot is enabled and no errors have occurred due to the build configuration.

Note: Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

Note: If the ESP32 is reset or powered down during the first boot, it will start the process again on the next boot.

9. On subsequent boots, the secure boot hardware will verify the software bootloader has not changed (using the secure bootloader key) and then the software bootloader will verify the signed partition table and app image (using the public key portion of the secure boot signing key).

4.24.6 Re-Flashable Software Bootloader

Configuration "Secure Boot: One-Time Flash" is the recommended configuration for production devices. In this mode, each device gets a unique key that is never stored outside the device.

However, an alternative mode *Secure Boot: Reflashable* is also available. This mode allows you to supply a binary key file that is used for the secure bootloader key. As you have the key file, you can generate new bootloader images and secure boot digests for them.

In the esp-idf build process, this 256-bit key file is derived from the ECDSA app signing key generated by the user (see the *Generating Secure Boot Signing Key* step below). This private key's SHA-256 digest is used as the secure bootloader key in efuse (as-is for Coding Scheme None, or truncate to 192 bytes for 3/4 Encoding). This is a convenience so you only need to generate/protect a single private key.

Note: Although it's possible, we strongly recommend not generating one secure boot key and flashing it to every device in a production environment. The "One-Time Flash" option is recommended for production environments.

To enable a reflashable bootloader:

1. In the *Project Configuration Menu*, select "Bootloader Config" -> *CONFIG_SECURE_BOOT* -> CONFIG_SECURE_BOOT_V1_ENABLED -> *CONFIG_SECURE_BOOTLOADER_MODE* -> Reflashable.
2. If necessary, set the *CONFIG_SECURE_BOOTLOADER_KEY_ENCODING* based on the coding scheme used by the device. The coding scheme is shown in the `Features` line when `esptool.py` connects to the chip, or in the `espefuse.py summary` output.
3. Please follow the steps shown in *Generating Secure Boot Signing Key* to generate signing key. Path of the generated key file must be specified in "Secure Boot Configuration" menu.
4. Run `idf.py bootloader`. A binary key file will be created, derived from the private key that is used for signing. Two sets of flashing steps will be printed - the first set of steps includes an `espefuse.py burn_key secure_boot_v1 path_to/secure-bootloader-key-xxx.bin` command which is used to write the bootloader key to efuse. (Flashing this key is a one-time-only process.) The second set of steps can be used to reflash the bootloader with a pre-calculated digest (generated during the build process).
5. Resume from *Step 6 of the one-time flashing process*, to flash the bootloader and enable secure boot. Watch the console log output closely to ensure there were no errors in the secure boot configuration.

4.24.7 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`. This uses the python-ecdsa library, which in turn uses Python's os.urandom() as a random number source.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available EC key generation utilities.

For example, to generate a signing key using the openssl command line:

` openssl ecparam -name prime256v1 -genkey -noout -out my_secure_boot_signing_key.pem `

Remember that the strength of the secure boot system depends on keeping the signing key private.

4.24.8 Remote Signing of Images

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default esp-idf secure boot configuration). The espsecure.py command line program can be used to sign app images & partition table data for secure boot, on a remote system.

To use remote signing, disable the option "Sign binaries during build". The private signing key does not need to be present on the build system. However, the public (signature verification) key is required because it is compiled into the bootloader (and can be used to verify image signatures during OTA updates.

To extract the public key from the private key:

```
espsecure.py extract_public_key --keyfile PRIVATE_SIGNING_KEY PUBLIC_VERIFICATION_
↪KEY
```

The path to the public signature verification key needs to be specified in the menuconfig under "Secure boot public signature verification key" in order to build the secure bootloader.

After the app image and partition table are built, the build system will print signing steps using espsecure.py:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY BINARY_FILE
```

The above command appends the image signature to the existing binary. You can use the *–output* argument to write the signed binary to a separate file:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY --output SIGNED_BINARY_FILE␣
↪BINARY_FILE
```

4.24.9 Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Keep the signing key private at all times. A leak of this key will compromise the secure boot system.
- Do not allow any third party to observe any aspects of the key generation or signing process using espsecure.py. Both processes are vulnerable to timing or other side-channel attacks.
- Enable all secure boot options in the Secure Boot Configuration. These include flash encryption, disabling of JTAG, disabling BASIC ROM interpeter, and disabling the UART bootloader encrypted flash access.
- Use secure boot in combination with *flash encryption* to prevent local readout of the flash contents.

4.24.10 Technical Details

The following sections contain low-level reference descriptions of various secure boot elements:

Secure Boot Hardware Support

The first stage of secure boot verification (checking the software bootloader) is done via hardware. The ESP32's Secure Boot support hardware can perform three basic operations:

1. Generate a random sequence of bytes from a hardware random number generator.
2. Generate a digest from data (usually the bootloader image from flash) using a key stored in Efuse block 2. The key in Efuse can (& should) be read/write protected, which prevents software access. For full details of this algorithm see *Secure Bootloader Digest Algorithm*. The digest can only be read back by software if Efuse ABS_DONE_0 is *not* burned (ie still 0).

3. Generate a digest from data (usually the bootloader image from flash) using the same algorithm as step 2 and compare it to a pre-calculated digest supplied in a buffer (usually read from flash offset 0x0). The hardware returns a true/false comparison without making the digest available to software. This function is available even when Efuse ABS_DONE_0 is burned.

Secure Bootloader Digest Algorithm

Starting with an "image" of binary data as input, this algorithm generates a digest as output. The digest is sometimes referred to as an "abstract" in hardware documentation.

For a Python version of this algorithm, see the `espsecure.py` tool in the components/esptool_py directory (specifically, the `digest_secure_bootloader` command).

Items marked with (^) are to fulfill hardware restrictions, as opposed to cryptographic restrictions.

1. Read the AES key from efuse block 2, in reversed byte order. If Coding Scheme is set to 3/4 Encoding, extend the 192 bit key to 256 bits using the same algorithm described in *Flash Encryption Algorithm*.
2. Prefix the image with a 128 byte randomly generated IV.
3. If the image length is not modulo 128, pad the image to a 128 byte boundary with 0xFF. (^)
4. For each 16 byte plaintext block of the input image: - Reverse the byte order of the plaintext input block (^) - Apply AES256 in ECB mode to the plaintext block. - Reverse the byte order of the ciphertext output block. (^) - Append to the overall ciphertext output.
5. Byte-swap each 4 byte word of the ciphertext (^)
6. Calculate SHA-512 of the ciphertext.

Output digest is 192 bytes of data: The 128 byte IV, followed by the 64 byte SHA-512 digest.

Image Signing Algorithm

Deterministic ECDSA as specified by RFC 6979.

- Curve is NIST256p (openssl calls this curve "prime256v1", it is also sometimes called secp256r1).
- Hash function is SHA256.
- Key format used for storage is PEM.
  - In the bootloader, the public key (for signature verification) is flashed as 64 raw bytes.
- Image signature is 68 bytes - a 4 byte version word (currently zero), followed by a 64 bytes of signature data. These 68 bytes are appended to an app image or partition table data.

Manual Commands

Secure boot is integrated into the esp-idf build system, so `make` will automatically sign an app image if secure boot is enabled. `idf.py bootloader` will produce a bootloader digest if menuconfig is configured for it.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --keyfile ./my_signing_key.pem --output ./image_signed.bin␣
↪image-unsigned.bin
```

Keyfile is the PEM file containing an ECDSA private signing key.

To generate a bootloader digest:

```
espsecure.py digest_secure_bootloader --keyfile ./securebootkey.bin --output ./
↪bootloader-digest.bin build/bootloader/bootloader.bin
```

Keyfile is the 32 byte raw secure boot key for the device.

The output of the `espsecure.py digest_secure_bootloader` command is a single file which contains both the digest and the bootloader appended to it. To flash the combined digest plus bootloader to the device:

```
esptool.py write_flash 0x0 bootloader-digest.bin
```

4.24.11 Secure Boot & Flash Encryption

If secure boot is used without *Flash Encryption*, it is possible to launch "time-of-check to time-of-use" attack, where flash contents are swapped after the image is verified and running. Therefore, it is recommended to use both the features together.

4.24.12 Signed App Verification Without Hardware Secure Boot

The integrity of apps can be checked even without enabling the hardware secure boot option. This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement. See *How To Enable Signed App Verification* for step by step instructions.

An app can be verified on update and, optionally, be verified on boot.

- Verification on update: When enabled, the signature is automatically checked whenever the esp_ota_ops.h APIs are used for OTA updates. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.
- Verification on boot: When enabled, the bootloader will be compiled with code to verify that an app is signed before booting it. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

How To Enable Signed App Verification

1. Open *Project Configuration Menu* -> Security features -> Enable *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*
2. "Bootloader verifies app signatures" can be enabled, which verifies app on boot.
3. By default, "Sign binaries during build" will be enabled on selecting "Require signed app images" option, which will sign binary files as a part of build process. The file named in "Secure boot private signing key" will be used to sign the image.
4. If you disable "Sign binaries during build" option then you'll have to enter path of a public key file used to verify signed images in "Secure boot public signature verification key". In this case, private signing key should be generated by following instructions in *Generating Secure Boot Signing Key*; public verification key and signed image should be generated by following instructions in *Remote Signing of Images*.

4.24.13 Advanced Features

JTAG Debugging

By default, when Secure Boot is enabled then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables Secure Boot.

See *JTAG with Flash Encryption or Secure Boot* for more information about using JTAG Debugging with either Secure Boot or signed app verification enabled.

4.25 Secure Boot V2

Important: This document is about Secure Boot V2, supported on the following chips: ESP32 (ECO3 onwards), ESP32-S2, ESP32-S3, ESP32-C3 (ECO3 onwards), and ESP32-C2. Except for ESP32, it is the only supported Secure Boot scheme.

For ESP32 before ECO3, refer to *Secure Boot*. It is recommended that users use Secure Boot V2 if they have a chip version that supports it. Secure Boot V2 is safer and more flexible than Secure Boot V1.

Secure Boot V2 uses RSA-PSS based app and bootloader verification. This document can also be used as a reference for signing apps using the RSA-PSS scheme without signing the bootloader.

`Secure Boot V2` and RSA scheme (`App Signing Scheme`) options are available for ESP32 from ECO3 onwards. To use these options in menuconfig, set *CONFIG_ESP32_REV_MIN* greater than or equal to *Rev 3*.

4.25.1 Background

Secure Boot protects a device from running any unauthorized (i.e., unsigned) code by checking that each piece of software that is being booted is signed. On an ESP32, these pieces of software include the second stage bootloader and each application binary. Note that the first stage bootloader does not require signing as it is ROM code thus cannot be changed.

The Secure Boot process on the ESP32 involves the following steps:

1. When the first stage bootloader loads the second stage bootloader, the second stage bootloader's RSA-PSS signature is verified. If the verification is successful, the second stage bootloader is executed.
2. When the second stage bootloader loads a particular application image, the application's RSA-PSS signature is verified. If the verification is successful, the application image is executed.

4.25.2 Advantages

- The RSA-PSS public key is stored on the device. The corresponding RSA-PSS private key is kept at a secret place and is never accessed by the device.
- Only one public key can be generated and stored in the chip during manufacturing.
- Same image format and signature verification method is applied for applications and software bootloader.
- No secrets are stored on the device. Therefore, it is immune to passive side-channel attacks (timing or power analysis, etc.)

4.25.3 Secure Boot V2 Process

This is an overview of the Secure Boot V2 Process. Instructions how to enable Secure Boot are supplied in section *How To Enable Secure Boot V2*.

Secure Boot V2 verifies the bootloader image and application binary images using a dedicated *signature block*. Each image has a separately generated signature block which is appended to the end of the image.

Only one signature block can be appended to the bootloader or application image in ESP32 ECO3.

Each signature block contains a signature of the preceding image as well as the corresponding RSA-3072 public key. For more details about the format, refer to *Signature Block Format*. A digest of the RSA-3072 public key is stored in the eFuse.

The application image is not only verified on every boot but also on each over the air (OTA) update. If the currently selected OTA app image cannot be verified, the bootloader will fall back and look for another correctly signed application image.

The Secure Boot V2 process follows these steps:

1. On startup, the ROM code checks the Secure Boot V2 bit in the eFuse. If Secure Boot is disabled, a normal boot will be executed. If Secure Boot is enabled, the boot will proceed according to the following steps.
2. The ROM code verifies the bootloader's signature block (*Verifying a Signature Block*). If this fails, the boot process will be aborted.
3. The ROM code verifies the bootloader image using the raw image data, its corresponding signature block(s), and the eFuse (*Verifying an Image*). If this fails, the boot process will be aborted.
4. The ROM code executes the bootloader.
5. The bootloader verifies the application image's signature block (*Verifying a Signature Block*). If this fails, the boot process will be aborted.
6. The bootloader verifies the application image using the raw image data, its corresponding signature blocks and the eFuse (*Verifying an Image*). If this fails, the boot process will be aborted. If the verification fails but another application image is found, the bootloader will then try to verify that other image using steps 5 to 7. This repeats until a valid image is found or no other images are found.
7. The bootloader executes the verified application image.

4.25.4 Signature Block Format

The signature block starts on a 4KB aligned boundary and has a flash sector of its own. The signature is calculated over all bytes in the image including the padding bytes (*Secure Padding*).

The content of each signature block is shown in the following table:

Table 28: Content of a Signature Block

Offset	Size (bytes)	Description
0	1	Magic byte
1	1	Version number byte (currently 0x02), 0x01 is for Secure Boot V1.
2	2	Padding bytes, Reserved. Should be zero.
4	32	SHA-256 hash of only the image content, not including the signature block.
36	384	RSA Public Modulus used for signature verification. (value 'n' in RFC8017).
420	4	RSA Public Exponent used for signature verification (value 'e' in RFC8017).
424	384	Pre-calculated R, derived from 'n'.
808	4	Pre-calculated M', derived from 'n'
812	384	RSA-PSS Signature result (section 8.1.1 of RFC8017) of image content, computed using following PSS parameters: SHA256 hash, MFG1 function, salt length 32 bytes, default trailer field (0xBC).
1196	4	CRC32 of the preceding 1196 bytes.
1200	16	Zero padding to length 1216 bytes.

Note: R and M' are used for hardware-assisted Montgomery Multiplication.

The remainder of the signature sector is erased flash (0xFF) which allows writing other signature blocks after previous signature block.

4.25.5 Secure Padding

In Secure Boot V2 scheme, the application image is padded to the flash MMU page size boundary to ensure that only verified contents are mapped in the internal address space. This is known as secure padding. Signature of the image is calculated after padding and then signature block (4KB) gets appended to the image.

- Default flash MMU page size is 64KB

- Secure padding is applied through the option `--secure-pad-v2` in the `elf2image` conversion using `esptool.py`

Following table explains the Secure Boot V2 signed image with secure padding and signature block appended:

Table 29: Contents of a signed application

Offset	Size (KB)	Description
0	580	Unsigned application size (as an example)
580	60	Secure padding (aligned to next 64KB boundary)
640	4	Signature block

Note: Please note that the application image always starts on the next flash MMU page size boundary (default 64KB) and hence the space left over after the signature block shown above can be utilized to store any other data partitions (e.g., `nvs`).

4.25.6 Verifying a Signature Block

A signature block is "valid" if the first byte is 0xe7 and a valid CRC32 is stored at offset 1196. Otherwise it's invalid.

4.25.7 Verifying an Image

An image is "verified" if the public key stored in any signature block is valid for this device, and if the stored signature is valid for the image data read from flash.

1. Compare the SHA-256 hash digest of the public key embedded in the bootloader's signature block with the digest(s) saved in the eFuses. If public key's hash doesn't match any of the hashes from the eFuses, the verification fails.
2. Generate the application image digest and match it with the image digest in the signature block. If the digests don't match, the verification fails.
3. Use the public key to verify the signature of the bootloader image, using RSA-PSS (section 8.1.2 of RFC8017) with the image digest calculated in step (2) for comparison.

4.25.8 Bootloader Size

Enabling Secure boot and/or flash encryption will increase the size of bootloader, which might require updating partition table offset. See *Bootloader Size*.

In the case when *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES* is disabled, the bootloader is sector padded (4KB) using the `--pad-to-size` option in `elf2image` command of `esptool`.

4.25.9 eFuse usage

ESP32-ECO3:

- ABS_DONE_1 - Enables Secure Boot protection on boot.
- BLK2 - Stores the SHA-256 digest of the public key. SHA-256 hash of public key modulus, exponent, pre-calculated R & M' values (represented as 776 bytes –offsets 36 to 812 - as per the *Signature Block Format*) is written to an eFuse key block. The write-protection bit must be set, but the read-protection bit must not.

The key(s) must be readable in order to give software access to it. If the key(s) is read-protected then the software reads the key(s) as all zeros and the signature verification process will fail, and the boot process will be aborted.

4.25.10 How To Enable Secure Boot V2

1. Open the *Project Configuration Menu*, in "Security features" set "Enable hardware Secure Boot in bootloader" to enable Secure Boot.
2. For ESP32, Secure Boot V2 is available only ESP32 ECO3 onwards. To view the "Secure Boot V2" option the chip revision should be changed to revision 3 (ESP32- ECO3). To change the chip revision, set "Minimum Supported ESP32 Revision" to Rev 3 in "Component Config" -> "ESP32- Specific".
3. Specify the path to Secure Boot signing key, relative to the project directory.
4. Select the desired UART ROM download mode in "UART ROM download mode". By default the UART ROM download mode has been kept enabled in order to prevent permanently disabling it in the development phase, this option is a potentially insecure option. It is recommended to disable the UART download mode for better security.
5. Set other menuconfig options (as desired). Then exit menuconfig and save your configuration.
6. The first time you run `idf.py build`, if the signing key is not found then an error message will be printed with a command to generate a signing key via `espsecure.py generate_signing_key`.

Important: A signing key generated this way will use the best random number source available to the OS and its Python installation (*/dev/urandom* on OSX/Linux and *CryptGenRandom()* on Windows). If this random number source is weak, then the private key will be weak.

Important: For production environments, we recommend generating the key pair using openssl or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.

---

7. Run `idf.py bootloader` to build a Secure Boot enabled bootloader. The build output will include a prompt for a flashing command, using `esptool.py write_flash`.
8. When you're ready to flash the bootloader, run the specified command (you have to enter it yourself, this step is not performed by the build system) and then wait for flashing to complete.
9. Run `idf.py flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 6.

Note: `idf.py flash` doesn't flash the bootloader if Secure Boot is enabled.

---

10. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable Secure Boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that Secure Boot is enabled and no errors have occurred due to the build configuration.

Note: Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

Note: If the ESP32 is reset or powered down during the first boot, it will start the process again on the next boot.

---

11. On subsequent boots, the Secure Boot hardware will verify the software bootloader has not changed and the software bootloader will verify the signed app image (using the validated public key portion of its appended signature block).

4.25.11 Restrictions after Secure Boot is enabled

- Any updated bootloader or app will need to be signed with a key matching the digest already stored in eFuse.

- After Secure Boot is enabled, no further eFuses can be read protected. (If *Flash Encryption* is enabled then the bootloader will ensure that any flash encryption key generated on first boot will already be read protected.) If *CONFIG_SECURE_BOOT_INSECURE* is enabled then this behavior can be disabled, but this is not recommended.

4.25.12 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`.

The `--version 2` parameter will generate the RSA 3072 private key for Secure Boot V2.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available RSA-PSS key generation utilities.

For example, to generate a signing key using the openssl command line:

```
` openssl genrsa -out my_secure_boot_signing_key.pem 3072 `
```

Remember that the strength of the Secure Boot system depends on keeping the signing key private.

4.25.13 Remote Signing of Images

Signing using espsecure.py

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default esp-idf Secure Boot configuration). The espsecure.py command line program can be used to sign app images & partition table data for Secure Boot, on a remote system.

To use remote signing, disable the option *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES* and build the firmware. The private signing key does not need to be present on the build system.

After the app image and partition table are built, the build system will print signing steps using espsecure.py:

```
espsecure.py sign_data BINARY_FILE --version 2 --keyfile PRIVATE_SIGNING_KEY
```

The above command appends the image signature to the existing binary. You can use the *–output* argument to write the signed binary to a separate file:

```
espsecure.py sign_data --version 2 --keyfile PRIVATE_SIGNING_KEY --output SIGNED_
↪BINARY_FILE BINARY_FILE
```

Signing using Pre-calculated Signatures

If you have valid pre-calculated signatures generated for an image and their corresponding public keys, you can use these signatures to generate a signature sector and append it to the image. Note that the pre-calculated signature should be calculated over all bytes in the image including the secure-padding bytes.

In such cases, the firmware image should be built by disabling the option *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES*. This image will be secure-padded and to generate a signed binary use the following command:

```
espsecure.py sign_data --version 2 --pub-key PUBLIC_SIGNING_KEY --signature␣
↪SIGNATURE_FILE --output SIGNED_BINARY_FILE BINARY_FILE
```

The above command verifies the signature, generates a signature block (refer to *Signature Block Format*) and appends it to the binary file.

Signing using an External Hardware Security Module (HSM)

For security reasons, you might also use an external Hardware Security Module (HSM) to store your private signing key, which cannot be accessed directly but has an interface to generate the signature of a binary file and its corresponding public key.

In such cases, disable the option *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES* and build the firmware. This secure-padded image then can be used to supply the external HSM for generating a signature. Refer to Signing using an External HSM to generate a signed image.

Note: For all the above three remote signing workflows, the signed binary is written to the filename provided to the `--output` argument.

4.25.14 Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Keep the signing key private at all times. A leak of this key will compromise the Secure Boot system.
- Do not allow any third party to observe any aspects of the key generation or signing process using espsecure.py. Both processes are vulnerable to timing or other side-channel attacks.
- Enable all Secure Boot options in the Secure Boot Configuration. These include flash encryption, disabling of JTAG, disabling BASIC ROM interpreter, and disabling the UART bootloader encrypted flash access.
- Use Secure Boot in combination with *flash encryption* to prevent local readout of the flash contents.

4.25.15 Technical Details

The following sections contain low-level reference descriptions of various Secure Boot elements:

Manual Commands

Secure boot is integrated into the esp-idf build system, so `idf.py build` will sign an app image and `idf.py bootloader` will produce a signed bootloader if secure signed binaries on build is enabled.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --version 2 --keyfile ./my_signing_key.pem --output ./image_
↪signed.bin image-unsigned.bin
```

Keyfile is the PEM file containing an RSA-3072 private signing key.

4.25.16 Secure Boot & Flash Encryption

If Secure Boot is used without *Flash Encryption*, it is possible to launch "time-of-check to time-of-use" attack, where flash contents are swapped after the image is verified and running. Therefore, it is recommended to use both the features together.

4.25.17 Signed App Verification Without Hardware Secure Boot

The Secure Boot V2 signature of apps can be checked on OTA update, without enabling the hardware Secure Boot option. This option uses the same app signature scheme as Secure Boot V2, but unlike hardware Secure Boot it does not prevent an attacker who can write to flash from bypassing the signature protection.

This may be desirable in cases where the delay of Secure Boot verification on startup is unacceptable, and/or where the threat model does not include physical access or attackers writing to bootloader or app partitions in flash.

In this mode, the public key which is present in the signature block of the currently running app will be used to verify the signature of a newly updated app. (The signature on the running app isn't verified during the update process, it's assumed to be valid.) In this way the system creates a chain of trust from the running app to the newly updated app.

For this reason, it's essential that the initial app flashed to the device is also signed. A check is run on app startup and the app will abort if no signatures are found. This is to try and prevent a situation where no update is possible. The app should have only one valid signature block in the first position. Note again that, unlike hardware Secure Boot V2, the signature of the running app isn't verified on boot. The system only verifies a signature block in the first position and ignores any other appended signatures.

Note: In general, it's recommended to use full hardware Secure Boot unless certain that this option is sufficient for application security needs.

How To Enable Signed App Verification

1. Open *Project Configuration Menu* -> Security features
2. Ensure *App Signing Scheme* is *RSA*. For ESP32 ECO3 chip, select *CONFIG_ESP32_REV_MIN* to *Rev 3* to get *RSA* option available
3. Enable *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*
4. By default, "Sign binaries during build" will be enabled on selecting "Require signed app images" option, which will sign binary files as a part of build process. The file named in "Secure boot private signing key" will be used to sign the image.
5. If you disable "Sign binaries during build" option then all app binaries must be manually signed by following instructions in *Remote Signing of Images*.

Warning: It is very important that all apps flashed have been signed, either during the build or after the build.

4.25.18 Advanced Features

JTAG Debugging

By default, when Secure Boot is enabled then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables Secure Boot.

See *JTAG with Flash Encryption or Secure Boot* for more information about using JTAG Debugging with either Secure Boot or signed app verification enabled.

4.26 Support for External RAM

4.26.1 Introduction

ESP32 has a few hundred kilobytes of internal RAM, residing on the same die as the rest of the chip components. It can be insufficient for some purposes, ESP32 has the ability to use up to 4 MB of virtual addresses for external SPI RAM memory. The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

4.26.2 Hardware

ESP32 supports SPI PSRAM (Psuedostatic RAM) connected in parallel with the SPI flash chip. While ESP32 is capable of supporting several types of RAM chips, ESP-IDF currently only supports Espressif branded PSRAM chips (e.g., ESP-PSRAM32, ESP-PSRAM64, etc).

Note: Some PSRAM chips are 1.8 V devices and some are 3.3 V. The working voltage of the PSRAM chip must match the working voltage of the flash component. Consult the datasheet for your PSRAM chip and ESP32 device to find out the working voltages. For a 1.8 V PSRAM chip, make sure to either set the MTDI pin to a high signal level on bootup, or program ESP32 eFuses to always use the VDD_SIO level of 1.8 V. Not doing this can damage the PSRAM and/or flash chip.

Note: Espressif produces both modules and system-in-package chips that integrate compatible PSRAM and flash and are ready to mount on a product PCB. Consult the Espressif website for more information. If you're using a custom PSRAM chip, ESP-IDF SDK might not be compatible with it.

For specific details about connecting the SoC or module pins to an external PSRAM chip, consult the SoC or module datasheet.

4.26.3 Configuring External RAM

ESP-IDF fully supports the use of external RAM in applications. Once the external RAM is initialized at startup, ESP-IDF can be configured to integrate the external RAM in several ways:

- *Integrate RAM into the ESP32 Memory Map*
- *Add External RAM to the Capability Allocator*
- *Provide External RAM via malloc()* (default)
- *Allow .bss Segment to be Placed in External Memory*
- *Allow .noinit Segment to be Placed in External Memory*

Integrate RAM into the ESP32 Memory Map

Select this option by choosing "Integrate RAM into memory map" from *CONFIG_SPIRAM_USE*.

This is the most basic option for external SPI RAM integration. Most likely, you will need another, more advanced option.

During the ESP-IDF startup, external RAM is mapped into the data address space, starting at address 0x3F800000 (byte-accessible). The length of this region is the same as the SPI RAM size (up to the limit of 4 MB).

Applications can manually place data in external memory by creating pointers to this region. So if an application uses external memory, it is responsible for all management of the external SPI RAM: coordinating buffer usage, preventing corruption, etc.

Add External RAM to the Capability Allocator

Select this option by choosing "Make RAM allocatable using heap_caps_malloc(…, MALLOC_CAP_SPIRAM)" from *CONFIG_SPIRAM_USE*.

When enabled, memory is mapped to address 0x3F800000 and also added to the *capabilities-based heap memory allocator* using `MALLOC_CAP_SPIRAM`.

To allocate memory from external RAM, a program should call `heap_caps_malloc(size, MALLOC_CAP_SPIRAM)`. After use, this memory can be freed by calling the normal `free()` function.

Provide External RAM via malloc()

Select this option by choosing "Make RAM allocatable using malloc() as well" from *CONFIG_SPIRAM_USE*. This is the default option.

In this case, memory is added to the capability allocator as described for the previous option. However, it is also added to the pool of RAM that can be returned by the standard `malloc()` function.

This allows any application to use the external RAM without having to rewrite the code to use `heap_caps_malloc(..., MALLOC_CAP_SPIRAM)`.

An additional configuration item, *CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL*, can be used to set the size threshold when a single allocation should prefer external memory:

- When allocating a size less than the threshold, the allocator will try internal memory first.
- When allocating a size equal to or larger than the threshold, the allocator will try external memory first.

If a suitable block of preferred internal/external memory is not available, the allocator will try the other type of memory.

Because some buffers can only be allocated in internal memory, a second configuration item *CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL* defines a pool of internal memory which is reserved for *only* explicitly internal allocations (such as memory for DMA use). Regular `malloc()` will not allocate from this pool. The *MALLOC_CAP_DMA* and `MALLOC_CAP_INTERNAL` flags can be used to allocate memory from this pool.

Allow .bss Segment to be Placed in External Memory

Enable this option by checking *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY*. This configuration setting is independent of the other three.

If enabled, the region of the data virtual address space where the PSRAM is mapped to will be used to store zero-initialized data (BSS segment) from the lwIP, net80211, libpp, wpa_supplicant and bluedroid ESP-IDF libraries.

Additional data can be moved from the internal BSS segment to external RAM by applying the macro `EXT_RAM_BSS_ATTR` to any static declaration (which is not initialized to a non-zero value).

It is also possible to place the BSS section of a component or a library to external RAM using linker fragment scheme `extram_bss`.

This option reduces the internal static memory used by the BSS segment.

Remaining external RAM can also be added to the capability heap allocator using the method shown above.

Allow .noinit Segment to be Placed in External Memory

Enable this option by checking *CONFIG_SPIRAM_ALLOW_NOINIT_SEG_EXTERNAL_MEMORY*. If enabled, a region of the address space provided in external RAM will be used to store non-initialized data. The values placed in this segment will not be initialized or modified even during startup or restart.

By applying the macro `EXT_RAM_NOINIT_ATTR`, data could be moved from the internal NOINIT segment to external RAM. Remaining external RAM can still be added to the capability heap allocator using the method shown above, *Add External RAM to the Capability Allocator*.

4.26.4 Restrictions

External RAM use has the following restrictions:

- When flash cache is disabled (for example, if the flash is being written to), the external RAM also becomes inaccessible; any reads from or writes to it will lead to an illegal cache access exception. This is also the reason why ESP-IDF does not by default allocate any task stacks in external RAM (see below).

- External RAM cannot be used as a place to store DMA transaction descriptors or as a buffer for a DMA transfer to read from or write into. Therefore when External RAM is enabled, any buffer that will be used in combination with DMA must be allocated using `heap_caps_malloc(size, MALLOC_CAP_DMA | MALLOC_CAP_INTERNAL)` and can be freed using a standard `free()` call.
- External RAM uses the same cache region as the external flash. This means that frequently accessed variables in external RAM can be read and modified almost as quickly as in internal ram. However, when accessing large chunks of data (>32 KB), the cache can be insufficient, and speeds will fall back to the access speed of the external RAM. Moreover, accessing large chunks of data can "push out" cached flash, possibly making the execution of code slower afterwards.
- In general, external RAM will not be used as task stack memory. *`xTaskCreate()`* and similar functions will always allocate internal memory for stack and task TCBs.

The option *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY* can be used to allow placing task stacks into external memory. In these cases *`xTaskCreateStatic()`* must be used to specify a task stack buffer allocated from external memory, otherwise task stacks will still be allocated from internal memory.

4.26.5 Failure to initialize

By default, failure to initialize external RAM will cause the ESP-IDF startup to abort. This can be disabled by enabling the config item *CONFIG_SPIRAM_IGNORE_NOTFOUND*.

If *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY* is enabled, the option to ignore failure is not available as the linker will have assigned symbols to external memory addresses at link time.

- Regarding stacks in PSRAM: For tasks that do not call ROM code in any way (directly or indirectly), the *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY* option will eliminate the check in *`xTaskCreateStatic()`*, allowing a task's stack to be in external RAM. However, using this is **not advised**.
- When used at 80 MHz clock speed, external RAM must also occupy either the HSPI or VSPI bus. Select which SPI host will be used by *CONFIG_SPIRAM_OCCUPY_SPI_HOST*.

4.26.6 Chip Revisions

There are some issues with certain revisions of ESP32 that have repercussions for use with external RAM. The issues are documented in the ESP32 Series SoC Errata document. In particular, ESP-IDF handles the bugs mentioned in the following ways:

ESP32 Rev v0.0

ESP-IDF has no workaround for the bugs in this revision of silicon, and it cannot be used to map external PSRAM into ESP32's main memory map.

ESP32 Rev v1.0

The bugs in this revision of silicon cause issues if certain sequences of machine instructions operate on external memory. (ESP32 Series SoC Errata 3.2). As a workaround, the `-mfix-esp32-psram-cache-issue` flag has been added to the ESP32 GCC compiler such that these sequences are filtered out. As a result, the compiler only outputs code that can safely be executed. The *CONFIG_SPIRAM_CACHE_WORKAROUND* option can be used to enable this workaround.

Aside from linking to a recompiled version of Newlib with the additional flag, ESP-IDF also does the following:

- Avoids using some ROM functions
- Allocates static memory for the Wi-Fi stack

ESP32 Rev v3.0

ESP32 rev v3.0 fixes the PSRAM cache issue found in rev v1.0. When *CONFIG_ESP32_REV_MIN* option is set to `rev v3.0`, compiler workarounds related to PSRAM will be disabled. For more information about ESP32 v3.0, see ESP32 Chip Revision v3.0 User Guide.

4.27 Thread Local Storage

4.27.1 Overview

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. ESP-IDF provides three ways to make use of such variables:

- *FreeRTOS Native API*: ESP-IDF FreeRTOS native API.
- *Pthread API*: ESP-IDF's pthread API.
- *C11 Standard*: C11 standard introduces special keyword to declare variables as thread local.

4.27.2 FreeRTOS Native API

The ESP-IDF FreeRTOS provides the following API to manage thread local variables:

- *`vTaskSetThreadLocalStoragePointer()`*
- *`pvTaskGetThreadLocalStoragePointer()`*
- *`vTaskSetThreadLocalStoragePointerAndDelCallback()`*

In this case maximum number of variables that can be allocated is limited by *CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS* configuration value. Variables are kept in the task control block (TCB) and accessed by their index. Note that index 0 is reserved for ESP-IDF internal uses.

Using that API user can allocate thread local variables of an arbitrary size and assign them to any number of tasks. Different tasks can have different sets of TLS variables.

If size of the variable is more then 4 bytes then user is responsible for allocating/deallocating memory for it. Variable's deallocation is initiated by FreeRTOS when task is deleted, but user must provide function (callback) to do proper cleanup.

4.27.3 Pthread API

The ESP-IDF provides the following *pthread API* to manage thread local variables:

- `pthread_key_create()`
- `pthread_key_delete()`
- `pthread_getspecific()`
- `pthread_setspecific()`

This API has all benefits of the one above, but eliminates some its limits. The number of variables is limited only by size of available memory on the heap. Due to the dynamic nature this API introduces additional performance overhead compared to the native one.

4.27.4 C11 Standard

The ESP-IDF FreeRTOS supports thread local variables according to C11 standard (ones specified with `__thread` keyword). For details on this GCC feature please see https://gcc.gnu.org/onlinedocs/gcc-5.5.0/gcc/Thread-Local.html#Thread-Local. Storage for that kind of variables is allocated on the task's stack. Note that area for all such variables in the program will be allocated on the stack of every task in the system even if that task does not use such variables at all. For example ESP-IDF system tasks (like `ipc`, `timer` tasks etc.) will also have that extra stack space allocated. So this feature should be used with care. There is a tradeoff: C11 thread local variables are quite handy

to use in programming and can be accessed using minimal CPU instructions, but this benefit goes with the cost of additional stack usage for all tasks in the system. Due to static nature of variables allocation all tasks in the system have the same sets of C11 thread local variables.

4.28 Tools

4.28.1 IDF Frontend - idf.py

The `idf.py` command-line tool provides a front-end for easily managing your project builds, deployment and debugging, and more. It manages several tools, for example:

- CMake, which configures the project to be built.
- Ninja, which builds the project.
- esptool.py, which flashes the target.

The *Step 5. First Steps on ESP-IDF* contains a brief introduction on how to set up `idf.py` to configure, build, and flash projects.

Important: `idf.py` should be run in an ESP-IDF project directory, i.e., a directory containing a `CMakeLists.txt` file. Older style projects that contain a `Makefile` will not work with `idf.py`.

Commands

Start a New Project: `create-project`

```
idf.py create-project <project name>
```

This command creates a new ESP-IDF project. Additionally, the folder where the project will be created in can be specified by the `--path` option.

Create a New Component: `create-component`

```
idf.py create-component <component name>
```

This command creates a new component, which will have a minimum set of files necessary for building. The `-C` option can be used to specify the directory the component will be created in. For more information about components see the *Component CMakeLists Files*.

Select the Target Chip: `set-target` ESP-IDF supports multiple targets (chips). A full list of supported targets in your version of ESP-IDF can be seen by running `idf.py --list-targets`.

```
idf.py set-target <target>
```

This command sets the current project target.

Important: `idf.py set-target` will clear the build directory and re-generate the `sdkconfig` file from scratch. The old `sdkconfig` file will be saved as `sdkconfig.old`.

Note: The behavior of the `idf.py set-target` command is equivalent to:

1. clearing the build directory (`idf.py fullclean`)
2. removing the sdkconfig file (`mv sdkconfig sdkconfig.old`)

3. configuring the project with the new target (`idf.py -DIDF_TARGET=esp32 reconfigure`)

It is also possible to pass the desired `IDF_TARGET` as an environment variable (e.g., `export IDF_TARGET=esp32s2`) or as a CMake variable (e.g., `-DIDF_TARGET=esp32s2` argument to CMake or `idf.py`). Setting the environment variable is a convenient method if you mostly work with one type of the chip.

To specify the default value of `IDF_TARGET` for a given project, please add the `CONFIG_IDF_TARGET` option to the project' s `sdkconfig.defaults` file, e.g., `CONFIG_IDF_TARGET="esp32s2"`. This value of the option will be used if `IDF_TARGET` is not specified by other methods, such as using an environment variable, a CMake variable, or the `idf.py set-target` command.

If the target has not been set by any of these methods, the build system will default to `esp32` target.

Start the Graphical Configuration Tool: `menuconfig`

```
idf.py menuconfig
```

Build the Project: `build`

```
idf.py build
```

This command builds the project found in the current directory. This can involve multiple steps:

- Create the build directory if needed. The sub-directory `build` is used to hold build output, although this can be changed with the `-B` option.
- Run CMake as necessary to configure the project and generate build files for the main build tool.
- Run the main build tool (Ninja or *GNU Make*). By default, the build tool is automatically detected but it can be explicitly set by passing the `-G` option to `idf.py`.

Building is incremental, so if no source files or configuration has changed since the last build, nothing will be done.

Additionally, the command can be run with `app`, `bootloader` and `partition-table` arguments to build only the app, bootloader or partition table as applicable.

Remove the Build Output: `clean`

```
idf.py clean
```

This command removes the project build output files from the build directory, and the project will be fully rebuilt on next build. Using this command does not remove the CMake configuration output inside the build folder.

Delete the Entire Build Contents: `fullclean`

```
idf.py fullclean
```

This command deletes the entire "build" directory contents, which includes all CMake configuration output. The next time the project is built, CMake will configure it from scratch. Note that this option recursively deletes **all** files in the build directory, so use with care. Project configuration is not deleted.

Flash the Project: `flash`

```
idf.py flash
```

This command automatically builds the project if necessary, and then flash it to the target. You can use `-p` and `-b` options to set serial port name and flasher baud rate, respectively.

Note: The environment variables `ESPPORT` and `ESPBAUD` can be used to set default values for the `-p` and `-b` options, respectively. Providing these options on the command line overrides the default.

Similarly to the `build` command, the command can be run with `app`, `bootloader` and `partition-table` arguments to flash only the app, bootloader or partition table as applicable.

Hints on How to Resolve Errors

`idf.py` will try to suggest hints on how to resolve errors. It works with a database of hints stored in tools/idf_py_actions/hints.yml and the hints will be printed if a match is found for the given error. The menuconfig, gdb and openocd targets are not supported at the moment by automatic hints on resolving errors.

The `--no-hints` argument of `idf.py` can be used to turn the hints off in case they are not desired.

Important Notes

Multiple `idf.py` commands can be combined into one. For example, `idf.py -p COM4 clean flash monitor` will clean the source tree, then build the project and flash it to the target before running the serial monitor.

The order of multiple `idf.py` commands on the same invocation is not important, as they will automatically be executed in the correct order for everything to take effect (e.g., building before flashing, erasing before flashing).

For commands that are not known to `idf.py`, an attempt to execute them as a build system target will be made.

The command `idf.py` supports shell autocompletion for bash, zsh and fish shells.

In order to make shell autocompletion supported, please make sure you have at least Python 3.5 and click 7.1 or newer (*Software*).

To enable autocompletion for `idf.py`, use the `export` command (*Step 4. Set up the environment variables*). Autocompletion is initiated by pressing the TAB key. Type `idf.py -` and press the TAB key to autocomplete options.

The autocomplete support for PowerShell is planned in the future.

Advanced Commands

Open the Documentation: `docs`

```
idf.py docs
```

This command opens the documentation for the projects target and ESP-IDF version in the browser.

Show Size: `size`

```
idf.py size
```

This command prints app size information including the occupied RAM and flash and section (i.e., .bss) sizes.

```
idf.py size
```

Similarly, this command prints the same information for each component used in the project.

```
idf.py size-files
```

This command prints size information per source file in the project.

If you define variable `-DOUTPUT_JSON=1` when running CMake (or `idf.py`), the output will be formatted as JSON not as human readable text. See `idf.py-size` for more information.

Reconfigure the Project: `reconfigure`

```
idf.py reconfigure
```

This command forces CMake to be rerun regardless of whether it is necessary. It's unnecessary during normal usage, but can be useful after adding/removing files from the source tree, or when modifying CMake cache variables. For example, `idf.py -DNAME='VALUE' reconfigure` can be used to set variable `NAME` in CMake cache to value `VALUE`.

Clean the Python Byte Code: `python-clean`

```
idf.py python-clean
```

This command deletes generated python byte code from the ESP-IDF directory. The byte code may cause issues when switching between ESP-IDF and Python versions. It is advised to run this target after switching versions of Python.

Generate a UF2 binary: `uf2`

```
idf.py uf2
```

This command will generate a UF2 (USB Flashing Format) binary `uf2.bin` in the build directory. This file includes all the necessary binaries (bootloader, app, and partition table) for flashing the target.

This UF2 file can be copied to a USB mass storage device exposed by another ESP running the ESP USB Bridge project. The bridge MCU will use it to flash the target MCU. This is as simple copying (or "drag-and-dropping") the file to the exposed disk accessed by a file explorer in your machine.

To generate a UF2 binary for the application only (not including the bootloader and partition table), use the `uf2-app` command.

```
idf.py uf2-app
```

Global Options

To list all available root level options, run `idf.py --help`. To list options that are specific for a subcommand, run `idf.py <command> --help`, e.g., `idf.py monitor --help`. Here is a list of some useful options:

- `-C <dir>` allows overriding the project directory from the default current working directory.
- `-B <dir>` allows overriding the build directory from the default `build` subdirectory of the project directory.
- `--ccache` enables CCache when compiling source files if the CCache tool is installed. This can dramatically reduce the build time.

Important: Note that some older versions of CCache may exhibit bugs on some platforms, so if files are not rebuilt as expected, try disabling CCache and rebuiling the project. To enable CCache by default, set the `IDF_CCACHE_ENABLE` environment variable to a non-zero value.

- `-v` flag causes both `idf.py` and the build system to produce verbose build output. This can be useful for debugging build problems.
- `--cmake-warn-uninitialized` (or `-w`) causes CMake to print uninitialized variable warnings found in the project directory only. This only controls CMake variable warnings inside CMake itself, not other types of build warnings. This option can also be set permanently by setting the `IDF_CMAKE_WARN_UNINITIALIZED` environment variable to a non-zero value.
- `--no-hints` flag disables hints on resolving errors and disable capturing output.

4.28.2 IDF Docker Image

IDF Docker image (`espressif/idf`) is intended for building applications and libraries with specific versions of ESP-IDF, when doing automated builds.

The image contains:

- Common utilities such as git, wget, curl, zip.
- Python 3.7 or newer.
- A copy of a specific version of ESP-IDF (see below for information about versions). `IDF_PATH` environment variable is set, and points to ESP-IDF location in the container.
- All the build tools required for the specific version of ESP-IDF: CMake, ninja, cross-compiler toolchains, etc.
- All Python packages required by ESP-IDF are installed in a virtual environment.

The image entrypoint sets up `PATH` environment variable to point to the correct version of tools, and activates the Python virtual environment. As a result, the environment is ready to use the ESP-IDF build system.

The image can also be used as a base for custom images, if additional utilities are required.

Tags

Multiple tags of this image are maintained:

- `latest`: tracks `master` branch of ESP-IDF
- `vX.Y`: corresponds to ESP-IDF release `vX.Y`
- `release-vX.Y`: tracks `release/vX.Y` branch of ESP-IDF

Note: Versions of ESP-IDF released before this feature was introduced do not have corresponding Docker image versions. You can check the up-to-date list of available tags at https://hub.docker.com/r/espressif/idf/tags.

Usage

Setting up Docker Before using the `espressif/idf` Docker image locally, make sure you have Docker installed. Follow the instructions at https://docs.docker.com/install/, if it is not installed yet.

If using the image in CI environment, consult the documentation of your CI service on how to specify the image used for the build process.

Building a project with CMake In the project directory, run:

```
docker run --rm -v $PWD:/project -w /project espressif/idf idf.py build
```

The above command explained:

- `docker run`: runs a Docker image. It is a shorter form of the command `docker container run`.
- `--rm`: removes the container when the build is finished
- `-v $PWD:/project`: mounts the current directory on the host (`$PWD`) as `/project` directory in the container
- `espressif/idf`: uses Docker image `espressif/idf` with tag `latest` (implicitly added by Docker when no tag is specified)
- `idf.py build`: runs this command inside the container

Note: When the mounted directory, `/project`, contains a git repository owned by a different user (`UID`) than the one running the Docker container, git commands executed within `/project` might fail, displaying an error message `fatal: detected dubious ownership in repository at '/project'`. To resolve this issue, you can designate the `/project` directory as safe by setting the IDF_GIT_SAFE_DIR environment variable

during the Docker container startup. For instance, you can achieve this by including `-e IDF_GIT_SAFE_DIR='/project'` as a parameter. Additionally, multiple directories can be specified by using a `:` separator. To entirely disable this git security check, `*` can be used.

To build with a specific docker image tag, specify it as `espressif/idf:TAG`, for example:

```
docker run --rm -v $PWD:/project -w /project espressif/idf:release-v4.4 idf.py␣
↪build
```

You can check the up-to-date list of available tags at https://hub.docker.com/r/espressif/idf/tags.

Using the image interactively It is also possible to do builds interactively, to debug build issues or test the automated build scripts. Start the container with *-i -t* flags:

```
docker run --rm -v $PWD:/project -w /project -it espressif/idf
```

Then inside the container, use `idf.py` as usual:

```
idf.py menuconfig
idf.py build
```

Note: Commands which communicate with the development board, such as `idf.py flash` and `idf.py monitor` will not work in the container unless the serial port is passed through into the container. However currently this is not possible with Docker for Windows (https://github.com/docker/for-win/issues/1018) and Docker for Mac (https://github.com/docker/for-mac/issues/900).

Building custom images

The Dockerfile in ESP-IDF repository provides several build arguments which can be used to customize the Docker image:

- `IDF_CLONE_URL`: URL of the repository to clone ESP-IDF from. Can be set to a custom URL when working with a fork of ESP-IDF. Default is `https://github.com/espressif/esp-idf.git`.
- `IDF_CLONE_BRANCH_OR_TAG`: Name of a git branch or tag use when cloning ESP-IDF. This value is passed to `git clone` command using the `--branch` argument. Default is `master`.
- `IDF_CHECKOUT_REF`: If this argument is set to a non-empty value, `git checkout $IDF_CHECKOUT_REF` command will be performed after cloning. This argument can be set to the SHA of the specific commit to check out, for example if some specific commit on a release branch is desired.
- `IDF_CLONE_SHALLOW`: If this argument is set to a non-empty value, `--depth=1 --shallow-submodules` arguments will be used when performing `git clone`. This significantly reduces the amount of data downloaded and the size of the resulting Docker image. However, if switching to a different branch in such a "shallow" repository is necessary, an additional `git fetch origin <branch>` command must be executed first.
- `IDF_INSTALL_TARGETS`: Comma-separated list of IDF targets to install toolchains for, or `all` to install toolchains for all targets. Selecting specific targets reduces the amount of data downloaded and the size of the resulting Docker image. Default is `all`.

To use these arguments, pass them via the `--build-arg` command line option. For example, the following command will build a Docker image with a shallow clone of ESP-IDF v4.4.1 and tools for ESP32-C3, only:

```
docker build -t idf-custom:v4.4.1-esp32c3 \
    --build-arg IDF_CLONE_BRANCH_OR_TAG=v4.4.1 \
    --build-arg IDF_CLONE_SHALLOW=1 \
    --build-arg IDF_INSTALL_TARGETS=esp32c3 \
    tools/docker
```

4.28.3 IDF Windows Installer

Command-line parameters

Windows Installer *esp-idf-tools-setup* provides the following command-line parameters:

- `/CONFIG=[PATH]` - Path to `ini` configuration file to override default configuration of the installer. Default: `config.ini`.
- `/GITCLEAN=[yes|no]` - Perform git clean and remove untracked directories in Offline mode installation. Default: yes.
- `/GITRECURSIVE=[yes|no]` - Clone recursively all git repository submodules. Default: yes
- `/GITREPO=[URL|PATH]` - URL of repository to clone ESP-IDF. Default: https://github.com/espressif/esp-idf.git
- `/GITRESET=[yes|no]` - Enable/Disable git reset of repository during installation. Default: yes.
- `/HELP` - Display command line options provided by Inno Setup installer.
- `/IDFDIR=[PATH]` - Path to directory where it will be installed. Default: `{userdesktop}\esp-idf}`
- `/IDFVERSION=[v4.3|v4.1|master]` - Use specific IDF version. E.g. v4.1, v4.2, master. Default: empty, pick the first version in the list.
- `/IDFVERSIONSURL=[URL]` - Use URL to download list of IDF versions. Default: https://dl.espressif.com/dl/esp-idf/idf_versions.txt
- `/LOG=[PATH]` - Store installation log file in specific directory. Default: empty.
- `/OFFLINE=[yes|no]` - Execute installation of Python packages by PIP in offline mode. The same result can be achieved by setting the environment variable PIP_NO_INDEX. Default: no.
- `/USEEMBEDDEDPYTHON=[yes|no]` - Use Embedded Python version for the installation. Set to `no` to allow Python selection screen in the installer. Default: yes.
- `/PYTHONNOUSERSITE=[yes|no]` - Set PYTHONNOUSERSITE variable before launching any Python command to avoid loading Python packages from AppDataRoaming. Default: yes.
- `/PYTHONWHEELSURL=[URL]` - Specify URLs to PyPi repositories for resolving binary Python Wheel dependencies. The same result can be achieved by setting the environment variable PIP_EXTRA_INDEX_URL. Default: https://dl.espressif.com/pypi
- `/SKIPSYSTEMCHECK=[yes|no]` - Skip System Check page. Default: no.
- `/VERYSILENT /SUPPRESSMSGBOXES /SP- /NOCANCEL` - Perform silent installation.

Unattended installation

The unattended installation of IDF can be achieved by following command-line parameters:

```
esp-idf-tools-setup-x.x.exe /VERYSILENT /SUPPRESSMSGBOXES /SP- /NOCANCEL
```

The installer detaches its process from the command-line. Waiting for installation to finish could be achieved by following PowerShell script:

```
esp-idf-tools-setup-x.x.exe /VERYSILENT /SUPPRESSMSGBOXES /SP- /NOCANCEL
$InstallerProcess = Get-Process esp-idf-tools-setup
Wait-Process -Id $InstallerProcess.id
```

Custom Python and custom location of Python wheels

The IDF installer is using by default embedded Python with reference to Python Wheel mirror.

Following parameters allows to select custom Python and custom location of Python wheels:

```
esp-idf-tools-setup-x.x.exe /USEEMBEDDEDPYTHON=no /PYTHONWHEELSURL=https://pypi.
↪org/simple/
```

4.28.4 IDF Component Manager

The IDF Component manager is a tool that downloads dependencies for any ESP-IDF CMake project. The download happens automatically during a run of CMake. It can source components either from the component registry or from a git repository.

A list of components can be found on https://components.espressif.com/

Using with a project

Dependencies for each component in the project are defined in a separate manifest file named `idf_component.yml` placed in the root of the component. The manifest file template can be created for a component by running `idf.py create-manifest --component=my_component`. When a new manifest is added to one of the components in the project it's necessary to reconfigure it manually by running `idf.py reconfigure`. Then build will track changes in `idf_component.yml` manifests and automatically triggers CMake when necessary.

There is an example application: example:*build_system/cmake/component_manager* that uses components installed by the component manager.

It's not necessary to have a manifest for components that don't need any managed dependencies.

When CMake configures the project (e.g. `idf.py reconfigure`) component manager does a few things:

- Processes `idf_component.yml` manifests for every component in the project and recursively solves dependencies
- Creates a `dependencies.lock` file in the root of the project with a full list of dependencies
- Downloads all dependencies to the `managed_components` directory

The lock-file `dependencies.lock` and content of `managed_components` directory is not supposed to be modified by a user. When the component manager runs it always make sure they are up to date. If these files were accidentally modified it's possible to re-run the component manager by triggering CMake with `idf.py reconfigure`

Defining dependencies in the manifest

```yaml
dependencies:
  # Required IDF version
  idf: ">=4.1"
  # Defining a dependency from the registry:
  # https://components.espressif.com/component/example/cmp
  example/cmp: ">=1.0.0"

  # # Other ways to define dependencies
  #
  # # For components maintained by Espressif only name can be used.
  # # Same as `espressif/cmp`
  # component: "~1.0.0"
  #
  # # Or in a longer form with extra parameters
  # component2:
  #   version: ">=2.0.0"
  #
  #   # For transient dependencies `public` flag can be set.
  #   # `public` flag doesn't affect the `main` component.
  #   # All dependencies of `main` are public by default.
  #   public: true
  #
  #   # For components hosted on non-default registry:
  #   service_url: "https://componentregistry.company.com"
  #
  # # For components in git repository:
```

(continues on next page)

(continued from previous page)

```
# test_component:
#   path: test_component
#   git: ssh://git@gitlab.com/user/components.git
#
# # For test projects during component development
# # components can be used from a local directory
# # with relative or absolute path
# some_local_component:
#   path: ../../projects/component
```

Disabling the Component Manager

The component manager can be explicitly disabled by setting `IDF_COMPONENT_MANAGER` environment variable to `0`.

4.28.5 IDF Clang Tidy

The IDF Clang Tidy is a tool that uses clang-tidy to run static analysis on your current app.

Warning: This functionality and the toolchain it relies on are still under development. There may be breaking changes before a final release.

Prerequisites

If you have never run this tool before, take the following steps to get this tool prepared.

1. Run the export scripts (`export.sh` / `export.bat` / ···) to set up the environment variables.
2. Run `pip install --upgrade pyclang` to install this plugin. The extra commands would be activated in `idf.py` automatically.
3. Run `idf_tools.py install xtensa-clang` to install the clang-tidy required binaries

   Note: This toolchain is still under development. After the final release, you don't have to install them manually.

4. Get file from the llvm repository and add the folder of this script to the `$PATH`. Or you could pass an optional argument `--run-clang-tidy-py` later when you call `idf.py clang-check`. Please don't forget to make the script executable.

   Note: This file would be bundled in future toolchain releases. This is a temporary workaround.

5. Run the export scripts (`export.sh` / `export.bat` / ···) again to refresh the environment variables.

Extra Commands

`clang-check` Run `idf.py clang-check` to re-generate the compilation database and run `clang-tidy` under your current project folder. The output would be written to `<project_dir>/warnings.txt`.

Run `idf.py clang-check --help` to see the full documentation.

`clang-html-report`

1. Run `pip install codereport` to install the additional dependency.

2. Run `idf.py clang-html-report` to generate an HTML report in folder `<project_dir>/html_report` according to the `warnings.txt`. Please open the `<project_dir>/html_report/index.html` in your browser to check the report.

Bug Report

This tool is hosted in espressif/clang-tidy-runner. If you faced any bugs or have any feature request, please report them via github issues.

4.28.6 Downloadable Tools

ESP-IDF build process relies on a number of tools: cross-compiler toolchains, CMake build system, and others.

Installing the tools using an OS-specific package manager (like apt, yum, brew, etc.) is the preferred method when the required version of the tool is available. This recommendation is reflected in the Getting Started guide. For example, on Linux and macOS it is recommended to install CMake using an OS package manager.

However, some of the tools are IDF-specific and are not available in OS package repositories. Furthermore, different versions of ESP-IDF require different versions of the tools to operate correctly. To solve these two problems, ESP-IDF provides a set of scripts for downloading and installing the correct versions of tools, and exposing them in the environment.

The rest of the document refers to these downloadable tools simply as "tools". Other kinds of tools used in ESP-IDF are:

- Python scripts bundled with ESP-IDF (such as `idf.py`)
- Python packages installed from PyPI.

The following sections explain the installation method, and provide the list of tools installed on each platform.

Note: This document is provided for advanced users who need to customize their installation, users who wish to understand the installation process, and ESP-IDF developers.

If you are looking for instructions on how to install the tools, see the *Getting Started Guide*.

Tools metadata file

The list of tools and tool versions required for each platform is located in tools/tools.json. The schema of this file is defined by tools/tools_schema.json.

This file is used by tools/idf_tools.py script when installing the tools or setting up the environment variables.

Tools installation directory

`IDF_TOOLS_PATH` environment variable specifies the location where the tools are to be downloaded and installed. If not set, `IDF_TOOLS_PATH` defaults to `HOME/.espressif` on Linux and macOS, and `%USER_PROFILE%\.espressif` on Windows.

Inside `IDF_TOOLS_PATH`, the scripts performing tools installation create the following directories and files:

- `dist` —where the archives of the tools are downloaded.
- `tools` —where the tools are extracted. The tools are extracted into subdirectories: `tools/TOOL_NAME/VERSION/`. This arrangement allows different versions of tools to be installed side by side.
- `idf-env.json` —user install options (targets, features) are stored in this file. Targets are selected chip targets for which tools are installed and kept up-to-date. Features determine the Python package set which should be installed. These options will be discussed later.

- `python_env` —not tools related; virtual Python environments are installed in the sub-directories. Note that the Python environment directory can be placed elsewhere by setting the `IDF_PYTHON_ENV_PATH` environment variable.
- `espidf.constraints.*.txt` —one constraint file for each ESP-IDF release containing Python package version requirements.

GitHub Assets Mirror

Most of the tools downloaded by the tools installer are GitHub Release Assets, which are files attached to a software release on GitHub.

If GitHub downloads are inaccessible or slow to access, it's possible to configure a GitHub assets mirror.

To use Espressif's download server, set the environment variable `IDF_GITHUB_ASSETS` to `dl.espressif.com/github_assets`. When the install process is downloading a tool from `github.com`, the URL will be rewritten to use this server instead.

Any mirror server can be used provided the URL matches the `github.com` download URL format: the install process will replace `https://github.com` with `https://${IDF_GITHUB_ASSETS}` for any GitHub asset URL that it downloads.

Note: The Espressif download server doesn't currently mirror everything from GitHub, it only mirrors files attached as Assets to some releases as well as source archives for some releases.

`idf_tools.py` script

tools/idf_tools.py script bundled with ESP-IDF performs several functions:

- `install`: Download the tool into `${IDF_TOOLS_PATH}/dist` directory, extract it into `${IDF_TOOLS_PATH}/tools/TOOL_NAME/VERSION`.
  `install` command accepts the list of tools to install, in `TOOL_NAME` or `TOOL_NAME@VERSION` format. If `all` is given, all the tools (required and optional ones) are installed. If no argument or `required` is given, only the required tools are installed.
- `download`: Similar to `install` but doesn't extract the tools. An optional `--platform` argument may be used to download the tools for the specific platform.
- `export`: Lists the environment variables which need to be set to use the installed tools. For most of the tools, setting `PATH` environment variable is sufficient, but some tools require extra environment variables.
  The environment variables can be listed in either of `shell` or `key-value` formats, set by `--format` parameter:
  - `export` optional parameters:
    - `--unset` Creates statement that unset some global variables, so the environment gets to the state it was before calling `export.{sh/fish}`.
    - `--add_paths_extras` Adds extra ESP-IDF-related paths of `$PATH` to `${IDF_TOOLS_PATH}/esp-idf.json`, which is used to remove global variables when the active ESP-IDF environment is deactivated. Example: While processing `export.{sh/fish}` script, new paths are added to global variable `$PATH`. This option is used to save these new paths to the `${IDF_TOOLS_PATH}/esp-idf.json`.
  - `shell` produces output suitable for evaluation in the shell. For example,

    ```
    export PATH="/home/user/.espressif/tools/tool/v1.0.0/bin:$PATH"
    ```

    on Linux and macOS, and

    ```
    set "PATH=C:\Users\user\.espressif\tools\v1.0.0\bin;%PATH%"
    ```

    on Windows.

Note: Exporting environment variables in Powershell format is not supported at the moment. `key-value` format may be used instead.

The output of this command may be used to update the environment variables, if the shell supports this. For example:

```
eval $($IDF_PATH/tools/idf_tools.py export)
```

- `key-value` produces output in *VARIABLE=VALUE* format, suitable for parsing by other scripts:

```
PATH=/home/user/.espressif/tools/tool/v1.0.0:$PATH
```

Note that the script consuming this output has to perform expansion of `$VAR` or `%VAR%` patterns found in the output.

- `list`: Lists the known versions of the tools, and indicates which ones are installed.
- `check`: For each tool, checks whether the tool is available in the system path and in `IDF_TOOLS_PATH`.
- `install-python-env`: Create a Python virtual environment in the `${IDF_TOOLS_PATH}/python_env` directory (or directly in the directory set by the `IDF_PYTHON_ENV_PATH` environment variable) and install there the required Python packages. An optional `--features` argument allows one to specify a comma-separated list of features to be added or removed. Feature that begins with `-` will be removed and features with `+` or without any sign will be added. Example syntax for removing feature `XY` is `--features=-XY` and for adding `--features=+XY` or `--features=XY`. If both removing and adding options are provided with the same feature, no operation is performed. For each feature a requirements file must exist. For example, feature `XY` is a valid feature if `${IDF_PATH}/tools/requirements/requirements.XY.txt` is an existing file with a list of Python packages to be installed. There is one mandatory `core` feature ensuring core functionality of ESP-IDF (build, flash, monitor, debug in console). There can be an arbitrary number of optional features. The selected list of features is stored in `idf-env.json`. The requirement files contain a list of the desired Python packages to be installed and `espidf.constraints.*.txt` downloaded from https://dl.espressif.com and stored in `${IDF_TOOLS_PATH}` the package version requirements for a given ESP-IDF version. Althought it is not recommended, the download and use of constraint files can be disabled with the `--no-constraints` argument or setting the `IDF_PYTHON_CHECK_CONSTRAINTS` environment variable to `no`.
- `check-python-dependencies`: Checks if all required Python packages are installed. Packages from `${IDF_PATH}/tools/requirements/requirements.*.txt` files selected by the feature list of `idf-env.json` are checked with the package versions specified in the `espidf.constraints.*.txt` file. The constraint file is downloaded with `install-python-env` command. The use of constraints files can be disabled similarly to the `install-python-env` command.
- `uninstall`: Print and remove tools, that are currently not used by active ESP-IDF version.
  - `--dry-run` Print installed unused tools.
  - `--remove-archives` Additionally remove all older versions of previously downloaded installation packages.

Install scripts

Shell-specific user-facing scripts are provided in the root of ESP-IDF repository to facilitate tools installation. These are:

- `install.bat` for Windows Command Prompt
- `install.ps1` for Powershell
- `install.sh` for Bash
- `install.fish` for Fish

Aside from downloading and installing the tools into `IDF_TOOLS_PATH`, these scripts prepare a Python virtual environment, and install the required packages into that environment.

These scripts accept optionally a comma separated list of chip targets and `--enable-*` arguments for enabling features. These arguments are passed to the `idf_tools.py` script which stores them in `idf-env.json`. Therefore, chip targets and features can be enabled incrementally.

Running the scripts without any optional arguments will install tools for all chip targets (by running `idf_tools.py`

`install --targets=all`) and Python packages for core ESP-IDF functionality (by running `idf_tools.py install-python-env --features=core`).

Or for example, `install.sh esp32` will install tools only for ESP32. See the *Getting Started Guide* for more examples.

`install.sh --enable-XY` will enable feature `XY` (by running `idf_tools.py install-python-env --features=core,XY`).

Export scripts

Since the installed tools are not permanently added into the user or system `PATH` environment variable, an extra step is required to use them in the command line. The following scripts modify the environment variables in the current shell to make the correct versions of the tools available:

- `export.bat` for Windows Command Prompt
- `export.ps1` for Powershell
- `export.sh` for Bash
- `export.fish` for Fish

Note: To modify the shell environment in Bash, `export.sh` must be "sourced" : `. ./export.sh` (note the leading dot and space).

`export.sh` may be used with shells other than Bash (such as zsh). However in this case the `IDF_PATH` environment variable must be set before running the script. When used in Bash, the script will guess the `IDF_PATH` value from its own location.

In addition to calling `idf_tools.py`, these scripts list the directories which have been added to the `PATH`.

Other installation methods

Depending on the environment, more user-friendly wrappers for `idf_tools.py` are provided:

- *IDF Tools installer for Windows* can download and install the tools. Internally the installer uses `idf_tools.py`.
- Eclipse Plugin includes a menu item to set up the tools. Internally the plugin calls `idf_tools.py`.
- VSCode Extension for ESP-IDF includes an onboarding flow. This flow helps setting up the tools. Although the extension does not rely on `idf_tools.py`, the same installation method is used.

Custom installation

Although the methods above are recommended for ESP-IDF users, they are not a must for building ESP-IDF applications. ESP-IDF build system expects that all the necessary tools are installed somewhere, and made available in the `PATH`.

Uninstall ESP-IDF

Uninstalling ESP-IDF requires removing both the tools and the environment variables that have been configured during the installation.

- Windows users using the *Windows ESP-IDF Tools Installer* can simply run the uninstall wizard to remove ESP-IDF.
- To remove an installation performed by running the supported *install scripts*, simply delete the *tools installation directory* including the downloaded and installed tools. Any environment variables set by the *export scripts* are not permanent and will not be present after opening a new environment.

- When dealing with a custom installation, in addition to deleting the tools as mentioned above, you may also need to manually revert any changes to environment variables or system paths that were made to accommodate the ESP-IDF tools (e.g., `IDF_PYTHON_ENV_PATH` or `IDF_TOOLS_PATH`). If you manually copied any tools, you would need to track and delete those files manually.
- If you installed any plugins like the ESP-IDF Eclipse Plugin or VSCode ESP-IDF Extension, you should follow the specific uninstallation instructions described in the documentation of those components.

Note: Uninstalling the ESP-IDF tools does not remove any project files or your code. Be mindful of what you are deleting to avoid losing any work. If you are unsure about a step, refer back to the installation instructions.

These instructions assume that the tools were installed following the procedures in this provided document. If you've used a custom installation method, you might need to adapt these instructions accordingly.

List of IDF Tools

xtensa-esp-elf-gdb GDB for Xtensa

License: GPL-3.0-or-later

More info: https://github.com/espressif/binutils-gdb

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-x86_64-linux-gnu.tar.gz SHA256: d0743ec43cd92c35452a9097f7863281de4e72f04120d63cfbcf9d591a373529
linux-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-aarch64-linux-gnu.tar.gz SHA256: bc1fac0366c6a08e26c45896ca21c8c90efc2cdd431b8ba084e8772e15502d0e
linux-armel	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-arm-linux-gnueabi.tar.gz SHA256: 25efc51d52b71f097ccec763c5c885c8f5026b432fec4b5badd6a5f36fe34d04
linux-armhf	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-arm-linux-gnueabihf.tar.gz SHA256: 0f9ff39fdec4d8c9c1ef33149a3fcdd2cf1bae121529c507817c994d5ac38ca4
linux-i686	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-i586-linux-gnu.tar.gz SHA256: e0af0b3b4a6b29a843cd5f47e331a966d9258f7d825b4656c6251490f71b05b2
macos	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-x86_64-apple-darwin14.tar.gz SHA256: bd146fd99a52b2d71c7ce0f62b9e18f3423d6cae7b2b2c954046b0dd7a23142f
macos-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-aarch64-apple-darwin21.1.tar.gz SHA256: 5edc76565bf9d2fadf24e443ddf3df7567354f336a65d4af5b2ee805cdfcec24
win32	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-i686-w64-mingw32.zip SHA256: ea4f3ee6b95ad1ad2e07108a21a50037a3e64a420cdeb34b2ba95d612faed898
win64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-x86_64-w64-mingw32.zip SHA256: 13bb97f39173948d1cfb6e651d9b335ea9d52f1fdd0dda1eda3a2d23d8c63644

riscv32-esp-elf-gdb GDB for RISC-V

License: GPL-3.0-or-later

More info: https://github.com/espressif/binutils-gdb

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-linux-gnu.tar.gz SHA256: 2c78b806be176b1e449e07ff83429d38dfc39a13f89a127ac1ffa6c1230537a0
linux-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-linux-gnu.tar.gz SHA256: 33f80117c8777aaff9179e27953e41764c5c46b3c576dc96a37ecc7a368807ec
linux-armel	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabi.tar.gz SHA256: 292e6ec0a9381c1480bbadf5caae25e86428b68fb5d030c9be7deda5e7f070e0
linux-armhf	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabihf.tar.gz SHA256: 3b803ab1ae619d62a885afd31c2798de77368d59b888c27ec6e525709e782ef5
linux-i686	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i586-linux-gnu.tar.gz SHA256: 68a25fbcfc6371ec4dbe503ec92211977eb2006f0c29e67dbce6b93c70c6b7ec
macos	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-apple-darwin14.tar.gz SHA256: 322c722e6c12225ed8cd97f95a0375105756dc5113d369958ce0858ad1a90257
macos-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-apple-darwin21.1.tar.gz SHA256: c2224b3a8d02451c530cf004c29653292d963a1b4021b4b472b862b6dbe97e0b
win32	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i686-w64-mingw32.zip SHA256: 4b42149a99dd87ee7e6dde25c99bad966c7f964253fa8f771593d7cef69f5602
win64	required	https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-w64-mingw32.zip SHA256: 728231546ad5006d34463f972658b2a89e52f660a42abab08a29bedd4a8046ad

xtensa-esp32-elf Toolchain for Xtensa (ESP32) based on GCC

License: GPL-3.0-with-GCC-exception

More info: https://github.com/espressif/crosstool-NG

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-linux-amd64.tar.xz SHA256: 698d8407e18275d18feb7d1afdb68800b97904fbe39080422fb8609afa49df30
linux-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-linux-arm64.tar.xz SHA256: 48ed01abff1e89e6fe1c3ebe4e00df6a0a67e53ae24979970464a4a3b64aa622
linux-armel	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-linux-armel.tar.xz SHA256: 0e6131a9ab4e3da0a153ee75097012823ccf21f90c69368c3bf53c8a086736f8
linux-armhf	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-linux-armhf.tar.xz SHA256: 74173665e228d8b1c988de0d743607a2f661e2bd24619c246e25dba7a01f46bd
linux-i686	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-linux-i686.tar.xz SHA256: d06511bb18057d72b555d6c5b62b0686f19e9f8c7d7eae218b712eed0907dbb2
macos	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-macos.tar.xz SHA256: 1c9d873c56469e3abec1e4214b7200d36804a605d4f0991e539b1577415409bf
macos-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-macos-arm64.tar.xz SHA256: 297249b0dc5307fd496c4d85d960b69824996c0c450a8c92f8414a5fd32a7c3b
win32	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-win32.zip SHA256: 858ee049d6d8de730ed3e30285c4adc1a9cdfe077b591ed0b6f2bfa5e3564f53
win64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32-elf-gcc11_2_0-esp-2022r1-win64.zip SHA256: f469aff6a71113e3a145466d814184339e02248b158357766970646f5d2a3da7

xtensa-esp32s2-elf Toolchain for Xtensa (ESP32-S2) based on GCC

License: GPL-3.0-with-GCC-exception

More info: https://github.com/espressif/crosstool-NG

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-linux-amd64.tar.xz SHA256: 56e5913b6662b8eec7d6b46780e668bc7e7cebef239e326a74f764c92a3cc841
linux-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-linux-arm64.tar.xz SHA256: 2f0ccc9d40279d6407ed9547250fb0434f16060faa94460c52b74614a38a1e21
linux-armel	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-linux-armel.tar.xz SHA256: f71974c4aaf3f637f6adaa28bbbdf3a911db3385e0ab1544844513ec65185cc5
linux-armhf	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-linux-armhf.tar.xz SHA256: 73e3be22c993f1112fcb1f7631d82552a6b759f82f12cfb78e669c7303d92b25
linux-i686	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-linux-i686.tar.xz SHA256: 504efe97ce24561537bd442494b1046fc8fb9cc43a1c06ef1afa4652b7517201
macos	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-macos.tar.xz SHA256: f53da9423490001727c5b6c3b8e1602b887783f0ed68e5defbb3c7712ada9631
macos-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-macos-arm64.tar.xz SHA256: 3592e0fbdb2ca438c7360d93fd62ef0e05ead2fc8144eff344bbe1971d333287
win32	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-win32.zip SHA256: 96b873210438713a84ea6e39e591cdbbeef453cb431d8392ac3fa2e68a48bc97
win64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s2-elf-gcc11_2_0-esp-2022r1-win64.zip SHA256: 9ab0387e08047916bbf7ff0d2eb974c710bcf2e042cb04037b4dd93c9186f676

xtensa-esp32s3-elf Toolchain for Xtensa (ESP32-S3) based on GCC

License: GPL-3.0-with-GCC-exception

More info: https://github.com/espressif/crosstool-NG

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-linux-amd64.tar.xz SHA256: 5058b2e724166c34ca09ec2d5377350252de8bce5039b06c00352f9a8151f76e
linux-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-linux-arm64.tar.xz SHA256: d2c6fb98a5018139a9f5af6eb808e968f1381a5b34547a185f4dec142b0fa44e
linux-armel	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-linux-armel.tar.xz SHA256: 9944e67d95a5de9875670c5cd5cb0bb282ebac235a38b5fd6d53069813fead9e
linux-armhf	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-linux-armhf.tar.xz SHA256: c0a8836dd709605f8d68ea1fd6e8ae79b3fa76274bfffdd8e79eeadc8f1f3ce1
linux-i686	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-linux-i686.tar.xz SHA256: 0feccf884e36b6e93c27c793729199b18df22a409557b16c90b2883a6748e041
macos	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-macos.tar.xz SHA256: 2b46730adc6afd8115e0be9365050a87f9523617e5e58ee35cb85ff1ddf2756c
macos-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-macos-arm64.tar.xz SHA256: bb449ac62b9917638b35234c98ce03ddf1cac75c2d80fbd67c46ecec08369838
win32	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-win32.zip SHA256: 0c9ec6d296b66523e3990b195b6597dfc4030f2335bf904b614f990ad6dabbde
win64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/xtensa-esp32s3-elf-gcc11_2_0-esp-2022r1-win64.zip SHA256: 7213a0bf22607e9c70febaabef37822c2ae5e071ac53d6467e6031b02bb0b2bf

xtensa-clang LLVM for Xtensa (ESP32, ESP32-S2) based on clang

License: Apache-2.0

More info: https://github.com/espressif/llvm-project

Platform	Required	Download
linux-amd64	optional	https://github.com/espressif/llvm-project/releases/download/esp-14.0.0-20220415/xtensa-esp32-elf-llvm14_0_0-esp-14.0.0-20220415-linux-amd64.tar.xz SHA256: b0148627912dacf4a4cab4596ba9467cb8dd771522ca27b9526bc57b88ff366f
macos	optional	https://github.com/espressif/llvm-project/releases/download/esp-14.0.0-20220415/xtensa-esp32-elf-llvm14_0_0-esp-14.0.0-20220415-macos.tar.xz SHA256: 1a78c598825ef168c0c5668aff7848825a7b9d014bffd1f2f2484ceea9df3841
macos-arm64	optional	https://github.com/espressif/llvm-project/releases/download/esp-14.0.0-20220415/xtensa-esp32-elf-llvm14_0_0-esp-14.0.0-20220415-macos.tar.xz SHA256: 1a78c598825ef168c0c5668aff7848825a7b9d014bffd1f2f2484ceea9df3841
win64	optional	https://github.com/espressif/llvm-project/releases/download/esp-14.0.0-20220415/xtensa-esp32-elf-llvm14_0_0-esp-14.0.0-20220415-win64.zip SHA256: 793e7bd9c40fcb9a4fababaaccf3e4c5b8d9554d406af1b1fbee51806bf8c5dd

riscv32-esp-elf Toolchain for 32-bit RISC-V based on GCC

License: GPL-3.0-with-GCC-exception

More info: https://github.com/espressif/crosstool-NG

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-linux-amd64.tar.xz SHA256: 52710f804df4a033a2b621cc16cfa21023b42052819a51e35a2a164140bbf665
linux-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-linux-arm64.tar.xz SHA256: 812a18f2ecdc3f72c1d098c4e8baa968841099ce9d9ecf95baea85ff71e11013
linux-armel	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-linux-armel.tar.xz SHA256: bc6e3ff8323d1f8b137374788b5615152281aab9e7561c55ab1504145677b6c7
linux-armhf	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-linux-armhf.tar.xz SHA256: 63f85a089fcd06939ed5e7e72ee5cdca590aa470075e409c0a4c59ef1cab3a7b
linux-i686	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-linux-i686.tar.xz SHA256: 39d7295c30a23b5ea91baf61c207718ce86d4b1589014b030e121300370f696d
macos	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-macos.tar.xz SHA256: d3a6f42b02a5f1485ba3fa92b8a9d9f307f643420e22b3765e88bbe4570aee01
macos-arm64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-macos-arm64.tar.xz SHA256: 23d9a715d932a3af57fd7393b0789f88d0f70fedaf5b803deb9ab81dee271bd6
win32	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-win32.zip SHA256: 3e677ef068d7f154d33b0d3788b5f985c5066d110028eac44e0f76b3bda4429b
win64	required	https://github.com/espressif/crosstool-NG/releases/download/esp-2022r1/riscv32-esp-elf-gcc11_2_0-esp-2022r1-win64.zip SHA256: 324a5c679fef75313766cc48d3433c48bf23985a11b5070c5d19144538c6357b

esp32ulp-elf Toolchain for ESP32 ULP coprocessor

License: GPL-3.0-or-later

More info: https://github.com/espressif/binutils-gdb

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-amd64.tar.gz SHA256: b1f7801c3a16162e72393ebb772c0cbfe4d22d907be7c2c2dac168736e9195fd
linux-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-arm64.tar.gz SHA256: d6671b31bab31b9b13aea25bb7d60f15484cb8bf961ddbf67a62867e5563eae5
linux-armel	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armel.tar.gz SHA256: e107e7a9cd50d630b034f435a16a52db5a57388dc639a99c4c393c5e429711e9
linux-armhf	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armhf.tar.gz SHA256: 6c6dd25477b2e758d4669da3774bf664d1f012442c880f17dfdf0339e9c3dae9
linux-i686	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-i686.tar.gz SHA256: beb9b6737c975369b6959007739c88f44eb5afbb220f40737071540b2c1a9064
macos	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos.tar.gz SHA256: 5a952087b621ced16af1e375feac1371a61cb51ab7e7b44cbefb5afda2d573de
macos-arm64	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos-arm64.tar.gz SHA256: 73bda8476ef92d4f4abee96519abbba40e5ee32f368427469447b83cc7bb9b42
win32	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win32.zip SHA256: 77344715ea7d7a7a9fd0b27653f880efaf3bcc1ac843f61492d8a0365d91f731
win64	required	https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win64.zip SHA256: 525e5b4c8299869a3fdddb51baad76612c5c104bd96952ae6460ad7e5b5a4e21

cmake CMake build system

On Linux and macOS, it is recommended to install CMake using the OS-specific package manager (like apt, yum, brew, etc.). However, for convenience it is possible to install CMake using idf_tools.py along with the other tools.

License: BSD-3-Clause

More info: https://github.com/Kitware/CMake

Platform	Required	Download
linux-amd64	optional	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-linux-x86_64.tar.gz SHA256: 726f88e6598523911e4bce9b059dc20b851aa77f97e4cc5573f4e42775a5c16f
linux-arm64	optional	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-linux-aarch64.tar.gz SHA256: 50c3b8e9d3a3cde850dd1ea143df9d1ae546cbc5e74dc6d223eefc1979189651
linux-armel	optional	https://dl.espressif.com/dl/cmake/cmake-3.24.0-Linux-armv7l.tar.gz SHA256: 7dc787ef968dfef92491a4f191b8739ff70f8a649608b811c7a737b52481beb0
linux-armhf	optional	https://dl.espressif.com/dl/cmake/cmake-3.24.0-Linux-armv7l.tar.gz SHA256: 7dc787ef968dfef92491a4f191b8739ff70f8a649608b811c7a737b52481beb0
macos	optional	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-macos-universal.tar.gz SHA256: 3e0cca74a56d9027dabb845a5a26e42ef8e8b33beb1655d6a724187a345145e4
macos-arm64	optional	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-macos-universal.tar.gz SHA256: 3e0cca74a56d9027dabb845a5a26e42ef8e8b33beb1655d6a724187a345145e4
win32	required	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-windows-x86_64.zip SHA256: b1ad8c2dbf0778e3efcc9fd61cd4a962e5c1af40aabdebee3d5074bcff2e103c
win64	required	https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-windows-x86_64.zip SHA256: b1ad8c2dbf0778e3efcc9fd61cd4a962e5c1af40aabdebee3d5074bcff2e103c

openocd-esp32 OpenOCD for ESP32

License: GPL-2.0-only

More info: https://github.com/espressif/openocd-esp32

Platform	Required	Download
linux-amd64	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-linux-amd64-0.12.0-esp32-20240318.tar.gz SHA256: cf26c5cef4f6b04aa23cd2778675604e5a74a4ce4d8d17b854d05fbcb782d52c
linux-arm64	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-linux-arm64-0.12.0-esp32-20240318.tar.gz SHA256: 9b97a37aa2cab94424a778c25c0b4aa0f90d6ef9cda764a1d9289d061305f4b7
linux-armel	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-linux-armel-0.12.0-esp32-20240318.tar.gz SHA256: b7e82776ec374983807d3389df09c632ad9bc8341f2075690b6b500319dfeaf4
linux-armhf	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-linux-armhf-0.12.0-esp32-20240318.tar.gz SHA256: 16f8f65f12e5ba034d328cda2567d6851a2aceb3c957d577f89401c2e1d3f93a
macos	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-macos-0.12.0-esp32-20240318.tar.gz SHA256: b16c3082c94df1079367c44d99f7a8605534cd48aabc18898e46e94a2c8c57e7
macos-arm64	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-macos-arm64-0.12.0-esp32-20240318.tar.gz SHA256: 534ec925ae6e35e869e4e4e6e4d2c4a1eb081f97ebcc2dd5efdc52d12f4c2f86
win32	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-win32-0.12.0-esp32-20240318.zip SHA256: d379329eba052435173ab0d69c9b15bc164a6ce489e2a67cd11169d2dabff633
win64	required	https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20240318/openocd-esp32-win32-0.12.0-esp32-20240318.zip SHA256: d379329eba052435173ab0d69c9b15bc164a6ce489e2a67cd11169d2dabff633

ninja Ninja build system

On Linux and macOS, it is recommended to install ninja using the OS-specific package manager (like apt, yum, brew, etc.). However, for convenience it is possible to install ninja using idf_tools.py along with the other tools.

License: Apache-2.0

More info: https://github.com/ninja-build/ninja

Platform	Required	Download
linux-amd64	optional	https://dl.espressif.com/dl/ninja-1.10.2-linux64.tar.gz SHA256: 32bb769de4d57aa7ee0e292cfcb7553e7cc8ea0961f7aa2b3aee60aa407c4033
macos	optional	https://dl.espressif.com/dl/ninja-1.10.2-osx.tar.gz SHA256: 847bb1ca4bc16d8dba6aeed3ecb5055498b86bc68c364c37583eb5738bb440f1
macos-arm64	optional	https://dl.espressif.com/dl/ninja-1.10.2-osx.tar.gz SHA256: 847bb1ca4bc16d8dba6aeed3ecb5055498b86bc68c364c37583eb5738bb440f1
win64	required	https://dl.espressif.com/dl/ninja-1.10.2-win64.zip SHA256: bbde850d247d2737c5764c927d1071cbb1f1957dcabda4a130fa8547c12c695f

idf-exe IDF wrapper tool for Windows

License: Apache-2.0

More info: https://github.com/espressif/idf_py_exe_tool

Platform	Required	Download
win32	required	https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip SHA256: 7c81ef534c562354a5402ab6b90a6eb1cc8473a9f4a7b7a7f93ebbd23b4a2755
win64	required	https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip SHA256: 7c81ef534c562354a5402ab6b90a6eb1cc8473a9f4a7b7a7f93ebbd23b4a2755

ccache Ccache (compiler cache)

License: GPL-3.0-or-later

More info: https://github.com/ccache/ccache

Platform	Required	Download
win64	required	https://github.com/ccache/ccache/releases/download/v4.6.2/ccache-4.6.2-windows-x86_64.zip SHA256: bf230b0936962eae43a3410d6477a7d0b9308e29f89a3091881d22e2502604c5

dfu-util dfu-util (Device Firmware Upgrade Utilities)

License: GPL-2.0-only

More info: http://dfu-util.sourceforge.net/

Platform	Required	Download
win64	required	https://dl.espressif.com/dl/dfu-util-0.9-win64.zip SHA256: 5816d7ec68ef3ac07b5ac9fb9837c57d2efe45b6a80a2f2bbe6b40b1c15c470e

esp-rom-elfs ESP ROM ELFs

License: Apache-2.0

More info: https://github.com/espressif/esp-rom-elfs

Platform	Required	Download
any	required	https://github.com/espressif/esp-rom-elfs/releases/download/20220823/esp-rom-elfs-20220823.tar.gz SHA256: add4bedbdd950c8409ff45bbf5610316e7d14c4635ea6906f057f2183ab3e3e9

4.29 Unit Testing in ESP32

ESP-IDF provides the following methods to test software.

- Target based tests using a central unit test application which runs on the esp32. These tests use the Unity unit test framework. They can be integrated into an ESP-IDF component by placing them in the component's `test` subdirectory. This document mainly introduces this target based tests.
- Linux-host based unit tests in which all the hardwares are abstracted via mocks. Linux-host based tests are still under development and only a small fraction of IDF components support them, currently. They are covered here: *Unit Testing on Linux*.

4.29.1 Normal Test Cases

Unit tests are located in the `test` subdirectory of a component. Tests are written in C, and a single C source file can contain multiple test cases. Test files start with the word "test".

Each test file should include the `unity.h` header and the header for the C module to be tested.

Tests are added in a function in the C file as follows:

```
TEST_CASE("test name", "[module name]")
{
        // Add test here
}
```

- The first argument is a descriptive name for the test.
- The second argument is an identifier in square brackets. Identifiers are used to group related test, or tests with specific properties.

Note: There is no need to add a main function with `UNITY_BEGIN()` and `UNITY_END()` in each test case. `unity_platform.c` will run `UNITY_BEGIN()` autonomously, and run the test cases, then call `UNITY_END()`.

The `test` subdirectory should contain a *component CMakeLists.txt*, since they are themselves components (i.e., a test component). ESP-IDF uses the Unity test framework located in the `unity` component. Thus, each test component should specify the `unity` component as a component requirement using the `REQUIRES` argument. Normally, components *should list their sources manually*; for component tests however, this requirement is relaxed and the use of the `SRC_DIRS` argument in `idf_component_register` is advised.

Overall, the minimal `test` subdirectory `CMakeLists.txt` file should contain the following:

```
idf_component_register(SRC_DIRS "."
                       INCLUDE_DIRS "."
                       REQUIRES unity)
```

See http://www.throwtheswitch.org/unity for more information about writing tests in Unity.

4.29.2 Multi-device Test Cases

The normal test cases will be executed on one DUT (Device Under Test). However, components that require some form of communication (e.g., GPIO, SPI) require another device to communicate with, thus cannot be tested through normal test cases. Multi-device test cases involve writing multiple test functions, and running them on multiple DUTs.

The following is an example of a multi-device test case:

```
void gpio_master_test()
{
    gpio_config_t slave_config = {
            .pin_bit_mask = 1 << MASTER_GPIO_PIN,
            .mode = GPIO_MODE_INPUT,
    };
    gpio_config(&slave_config);
    unity_wait_for_signal("output high level");
    TEST_ASSERT(gpio_get_level(MASTER_GPIO_PIN) == 1);
}

void gpio_slave_test()
{
    gpio_config_t master_config = {
            .pin_bit_mask = 1 << SLAVE_GPIO_PIN,
            .mode = GPIO_MODE_OUTPUT,
    };
    gpio_config(&master_config);
    gpio_set_level(SLAVE_GPIO_PIN, 1);
    unity_send_signal("output high level");
}

TEST_CASE_MULTIPLE_DEVICES("gpio multiple devices test example", "[driver]", gpio_
↪master_test, gpio_slave_test);
```

The macro `TEST_CASE_MULTIPLE_DEVICES` is used to declare a multi-device test case.

- The first argument is test case name.
- The second argument is test case description.
- From the third argument, up to 5 test functions can be defined, each function will be the entry point of tests running on each DUT.

Running test cases from different DUTs could require synchronizing between DUTs. We provide `unity_wait_for_signal` and `unity_send_signal` to support synchronizing with UART. As the scenario in the above example, the slave should get GPIO level after master set level. DUT UART console will prompt and user interaction is required:

DUT1 (master) console:

```
Waiting for signal: [output high level]!
Please press "Enter" key to once any board send this signal.
```

DUT2 (slave) console:

```
Send signal: [output high level]!
```

Once the signal is sent from DUT2, you need to press "Enter" on DUT1, then DUT1 unblocks from `unity_wait_for_signal` and starts to change GPIO level.

4.29.3 Multi-stage Test Cases

The normal test cases are expected to finish without reset (or only need to check if reset happens). Sometimes we expect to run some specific tests after certain kinds of reset. For example, we want to test if the reset reason is correct after a wake up from deep sleep. We need to create a deep-sleep reset first and then check the reset reason. To support this, we can define multi-stage test cases, to group a set of test functions:

```
static void trigger_deepsleep(void)
{
    esp_sleep_enable_timer_wakeup(2000);
    esp_deep_sleep_start();
}

void check_deepsleep_reset_reason()
{
    soc_reset_reason_t reason = esp_rom_get_reset_reason(0);
    TEST_ASSERT(reason == RESET_REASON_CORE_DEEP_SLEEP);
}

TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32]", trigger_
↪deepsleep, check_deepsleep_reset_reason);
```

Multi-stage test cases present a group of test functions to users. It needs user interactions (select cases and select different stages) to run the case.

4.29.4 Tests For Different Targets

Some tests (especially those related to hardware) cannot run on all targets. Below is a guide how to make your unit tests run on only specified targets.

1. Wrap your test code by `!(TEMPORARY_)DISABLED_FOR_TARGETS()` macros and place them either in the original test file, or separate the code into files grouped by functions, but make sure all these files will be processed by the compiler. E.g.:

   ```
   #if !TEMPORARY_DISABLED_FOR_TARGETS(ESP32, ESP8266)
   TEST_CASE("a test that is not ready for esp32 and esp8266 yet", "[]")
   {
   }
   #endif //!TEMPORARY_DISABLED_FOR_TARGETS(ESP32, ESP8266)
   ```

Once you need one of the tests to be compiled on a specified target, just modify the targets in the disabled list. It's more encouraged to use some general conception that can be described in `soc_caps.h` to control the disabling of tests. If this is done but some of the tests are not ready yet, use both of them (and remove `!(TEMPORARY_)DISABLED_FOR_TARGETS()` later). E.g.:

```
#if SOC_SDIO_SLAVE_SUPPORTED
#if !TEMPORARY_DISABLED_FOR_TARGETS(ESP64)
TEST_CASE("a sdio slave tests that is not ready for esp64 yet", "[sdio_slave]")
{
    //available for esp32 now, and will be available for esp64 in the future
}
#endif //!TEMPORARY_DISABLED_FOR_TARGETS(ESP64)
#endif //SOC_SDIO_SLAVE_SUPPORTED
```

2. For test code that you are 100% for sure that will not be supported (e.g. no peripheral at all), use `DISABLED_FOR_TARGETS`; for test code that should be disabled temporarily, or due to lack of runners, etc., use `TEMPORARY_DISABLED_FOR_TARGETS`.

Some old ways of disabling unit tests for targets, that have obvious disadvantages, are deprecated:

- DON'T put the test code under `test/target` folder and use CMakeLists.txt to choose one of the target folder. This is prevented because test code is more likely to be reused than the implementations. If you put

something into `test/esp32` just to avoid building it on esp32s2, it's hard to make the code tidy if you want to enable the test again on esp32s3.

- DON'T use `CONFIG_IDF_TARGET_xxx` macros to disable the test items any more. This makes it harder to track disabled tests and enable them again. Also, a black-list style `#if !disabled` is preferred to white-list style `#if CONFIG_IDF_TARGET_xxx`, since you will not silently disable cases when new targets are added in the future. But for test implementations, it's allowed to use `#if CONFIG_IDF_TARGET_xxx` to pick one of the implementation code.
  - Test item: some items that will be performed on some targets, but skipped on other targets. E.g. There are three test items SD 1-bit, SD 4-bit and SDSPI. For ESP32-S2, which doesn't have SD host, among the tests only SDSPI is enabled on ESP32-S2.
  - Test implementation: some code will always happen, but in different ways. E.g. There is no SDIO PKT_LEN register on ESP8266. If you want to get the length from the slave as a step in the test process, you can have different implementation code protected by `#if CONFIG_IDF_TARGET_` reading in different ways.
    But please avoid using `#else` macro. When new target is added, the test case will fail at building stage, so that the maintainer will be aware of this, and choose one of the implementations explicitly.

4.29.5 Building Unit Test App

Follow the setup instructions in the top-level esp-idf README. Make sure that `IDF_PATH` environment variable is set to point to the path of esp-idf top-level directory.

Change into `tools/unit-test-app` directory to configure and build it:

- `idf.py menuconfig` - configure unit test app.
- `idf.py -T all build` - build unit test app with tests for each component having tests in the `test` subdirectory.
- `idf.py -T "xxx yyy" build` - build unit test app with tests for some space-separated specific components (For instance: `idf.py -T heap build` - build unit tests only for `heap` component directory).
- `idf.py -T all -E "xxx yyy" build` - build unit test app with all unit tests, except for unit tests of some components (For instance: `idf.py -T all -E "ulp mbedtls" build` - build all unit tests excludes `ulp` and `mbedtls` components).

Note: Due to inherent limitations of Windows command prompt, following syntax has to be used in order to build unit-test-app with multiple components: `idf.py -T xxx -T yyy build` or with escaped quotes: `idf.py -T \`"xxx yyy\`" build` in PowerShell or `idf.py -T \^"ssd1306 hts221\^" build` in Windows command prompt.

When the build finishes, it will print instructions for flashing the chip. You can simply run `idf.py flash` to flash all build output.

You can also run `idf.py -T all flash` or `idf.py -T xxx flash` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use menuconfig to set the serial port for flashing. For more information, see tools/unit-test-app/README.md.

4.29.6 Running Unit Tests

After flashing reset the ESP32 and it will boot the unit test app.

When unit test app is idle, press "Enter" will make it print test menu with all available tests:

```
Here's the test menu, pick your combo:
(1)     "esp_ota_begin() verifies arguments" [ota]
(2)     "esp_ota_get_next_update_partition logic" [ota]
(3)     "Verify bootloader image in flash" [bootloader_support]
(4)     "Verify unit test app image" [bootloader_support]
(5)     "can use new and delete" [cxx]
```

(continues on next page)

(continued from previous page)

```
(6)     "can call virtual functions" [cxx]
(7)     "can use static initializers for non-POD types" [cxx]
(8)     "can use std::vector" [cxx]
(9)     "static initialization guards work as expected" [cxx]
(10)    "global initializers run in the correct order" [cxx]
(11)    "before scheduler has started, static initializers work correctly" [cxx]
(12)    "adc2 work with wifi" [adc]
(13)    "gpio master/slave test example" [ignore][misc][test_env=UT_T2_1][multi_
↪device]
        (1)     "gpio_master_test"
        (2)     "gpio_slave_test"
(14)    "SPI Master clockdiv calculation routines" [spi]
(15)    "SPI Master test" [spi][ignore]
(16)    "SPI Master test, interaction of multiple devs" [spi][ignore]
(17)    "SPI Master no response when switch from host1 (SPI2) to host2 (SPI3)"␣
↪[spi]
(18)    "SPI Master DMA test, TX and RX in different regions" [spi]
(19)    "SPI Master DMA test: length, start, not aligned" [spi]
(20)    "reset reason check for deepsleep" [esp32][test_env=UT_T2_1][multi_stage]
        (1)     "trigger_deepsleep"
        (2)     "check_deepsleep_reset_reason"
```

The normal case will print the case name and description. Master-slave cases will also print the sub-menu (the registered test function names).

Test cases can be run by inputting one of the following:

- Test case name in quotation marks to run a single test case
- Test case index to run a single test case
- Module name in square brackets to run all test cases for a specific module
- An asterisk to run all test cases

`[multi_device]` and `[multi_stage]` tags tell the test runner whether a test case is a multiple devices or multiple stages of test case. These tags are automatically added by `` `TEST_CASE_MULTIPLE_STAGES `` and `TEST_CASE_MULTIPLE_DEVICES` macros.

After you select a multi-device test case, it will print sub-menu:

```
Running gpio master/slave test example...
gpio master/slave test example
        (1)     "gpio_master_test"
        (2)     "gpio_slave_test"
```

You need to input a number to select the test running on the DUT.

Similar to multi-device test cases, multi-stage test cases will also print sub-menu:

```
Running reset reason check for deepsleep...
reset reason check for deepsleep
        (1)     "trigger_deepsleep"
        (2)     "check_deepsleep_reset_reason"
```

First time you execute this case, input `1` to run first stage (trigger deepsleep). After DUT is rebooted and able to run test cases, select this case again and input `2` to run the second stage. The case only passes if the last stage passes and all previous stages trigger reset.

4.29.7 Timing Code with Cache Compensated Timer

Instructions and data stored in external memory (e.g. SPI Flash and SPI RAM) are accessed through the CPU's unified instruction and data cache. When code or data is in cache, access is very fast (i.e., a cache hit).

However, if the instruction or data is not in cache, it needs to be fetched from external memory (i.e., a cache miss). Access to external memory is significantly slower, as the CPU must execute stall cycles whilst waiting for the instruction or data to be retrieved from external memory. This can cause the overall code execution speed to vary depending on the number of cache hits or misses.

Code and data placements can vary between builds, and some arrangements may be more favorable with regards to cache access (i.e., minimizing cache misses). This can technically affect execution speed, however these factors are usually irrelevant as their effect 'average out' over the device's operation.

The effect of the cache on execution speed, however, can be relevant in benchmarking scenarios (especially micro benchmarks). There might be some variability in measured time between runs and between different builds. A technique for eliminating for some of the variability is to place code and data in instruction or data RAM (IRAM/DRAM), respectively. The CPU can access IRAM and DRAM directly, eliminating the cache out of the equation. However, this might not always be viable as the size of IRAM and DRAM is limited.

The cache compensated timer is an alternative to placing the code/data to be benchmarked in IRAM/DRAM. This timer uses the processor's internal event counters in order to determine the amount of time spent on waiting for code/data in case of a cache miss, then subtract that from the recorded wall time.

```
// Start the timer
ccomp_timer_start();

// Function to time
func_code_to_time();

// Stop the timer, and return the elapsed time in microseconds relative to
// ccomp_timer_start
int64_t t = ccomp_timer_stop();
```

One limitation of the cache compensated timer is that the task that benchmarked functions should be pinned to a core. This is due to each core having its own event counters that are independent of each other. For example, if `ccomp_timer_start` gets called on one core, put to sleep by the scheduler, wakes up, and gets rescheduled on the other core, then the corresponding `ccomp_timer_stop` will be invalid.

4.29.8 Mocks

Note: Currently, mocking is only possible with some selected components when running on the Linux host. In the future, we plan to make essential components in IDF mockable. This will also include mocking when running on the ESP32.

One of the biggest problems regarding unit testing of embedded systems are the strong hardware dependencies. Running unit tests directly on the ESP32 can be especially difficult for higher layer components for the following reasons:

- Decreased test reliability due to lower layer components and/or hardware setup.
- Increased difficulty in testing edge cases due to limitations of lower layer components and/or hardware setup
- Increased difficulty in identifying the root cause due to the large number of dependencies influencing the behavior

When testing a particular component, (i.e., the component under test), software mocking allows the dependencies of the component under test to be substituted (i.e., mocked) entirely in software. To allow software mocking, ESP-IDF integrates the CMock mocking framework as a component. With the addition of some CMake functions in the ESP-IDF's build system, it is possible to conveniently mock the entirety (or a part of) an IDF component.

Ideally, all components that the component under test is dependent on should be mocked, thus allowing the test environment complete control over all interactions with the component under test. However, if mocking all dependent components becomes too complex or too tedious (e.g. because you need to mock too many function calls) you have the following options:

- Include more "real" IDF code in the tests. This may work but increases the dependency on the "real" code's behavior. Furthermore, once a test fails, you may not know if the failure is in your actual code under tests or the "real" IDF code.
- Re-evaluate the design of the code under test and attempt to reduce its dependencies by dividing the code under test into more manageable components. This may seem burdensome but it is common knowledge that unit tests often expose software design weaknesses. Fixing design weaknesses will not only help with unit testing in the short term, but will help future code maintenance as well.

Refer to cmock/CMock/docs/CMock_Summary.md for more details on how CMock works and how to create and use mocks.

Requirements

The Linux target is the only target where mocking currently works. The following requirements are necessary to generate the mocks:

- Installed ESP-IDF including all ESP-IDF requirements
- System package requirements (`libbsd`, `libbsd-dev`)
- A recent enough Linux or macOS version and GCC compiler
- All components the application depends on must be either supported on the Linux target (Linux/POSIX simulator) or mock-able

An application that runs on the Linux target has to set the `COMPONENTS` variable to `main` in the CMakeLists.txt of the application's root directory:

```
set(COMPONENTS main)
```

This prevents the automatic inclusion of all components from ESP-IDF to the build process which is otherwise done for convenience.

Mock a Component

To create a mock version of a component, called a *component mock*, the component needs to be overwritten in a particular way. Overriding a component entails creating a component with the exact same name as the original component, then let the build system discover it later than the original component (see *Multiple components with the same name* for more details).

In the component mock, the following parts are specified:

- The headers providing the functions to generate mocks for
- Include paths of the aforementioned headers
- Dependencies of the mock component (this is necessary e.g. if the headers include files from other components)

All these parts have to be specified using the IDF build system function `idf_component_mock`. You can use the IDF build system function `idf_component_get_property` with the tag `COMPONENT_OVERRIDEN_DIR` to access the component directory of the original component and then register the mock component parts using `idf_component_mock`:

```
idf_component_get_property(original_component_dir <original-component-name>␣
↪COMPONENT_OVERRIDEN_DIR)
...
idf_component_mock(INCLUDE_DIRS "${original_component_dir}/include"
    REQUIRES freertos
    MOCK_HEADER_FILES ${original_component_dir}/include/header_containing_
↪functions_to_mock.h)
```

The component mock also requires a separate `mock` directory containing a `mock_config.yaml` file that configures CMock. A simple `mock_config.yaml` could look like this:

```
:cmock:
  :plugins:
    - expect
    - expect_any_args
```

For more details about the CMock configuration yaml file, have a look at cmock/CMock/docs/CMock_Summary.md.

Note that the component mock does not have to mock the original component in its entirety. As long as the test project's dependencies and dependencies of other code to the original components are satisfied by the component mock, partial mocking is adequate. In fact, most of the component mocks in IDF in `tools/mocks` are only partially mocking the original component.

Examples of component mocks can be found under tools/mocks in the IDF directory. General information on how to *override an IDF component* can be found in *Multiple components with the same name*.

Adjustments in Unit Test

The unit test needs to inform the cmake build system to mock dependent components (i.e., it needs to override the original component with the mock component). This is done by either placing the component mock into the project's `components` directory or adding the mock component's directory using the following line in the project's root `CMakeLists.txt`:

```
list(APPEND EXTRA_COMPONENT_DIRS "<mock_component_dir>")
```

Both methods will override existing components in ESP-IDF with the component mock. The latter is particularly convenient if you use component mocks that are already supplied by IDF.

Users should refer to the `esp_event` host-based unit test and its esp_event/host_test/esp_event_unit_test/CMakeLists.txt as an example of a component mock.

4.30 Unit Testing on Linux

Note: Host testing with IDF is experimental for now. We try our best to keep interfaces stable but can't guarantee it for now. Feedback via github or the forum on esp32.com is highly welcome, though and may influence the future design of the host-based tests.

This article provides an overview of unit tests with IDF on Linux. For using unit tests on the target, please refer to *target based unit testing*.

4.30.1 Embedded Software Tests

Embedded software tests are challenging due to the following factors:

- Difficulties running tests efficiently.
- Lack of many operating system abstractions when interfacing with hardware, making it difficult to isolate code under test.

To solve these two problems, Linux host-based tests with CMock are introduced. Linux host-based tests are more efficient than unit tests on the target since they:

- Compile the necessary code only
- Don't need time to upload to a target
- Run much faster on a host-computer, compared to an ESP

Using the CMock framework also solves the problem of hardware dependencies. Through mocking, hardware details are emulated and specified at run time, but only if necessary.

Of course, using code on the host and using mocks does not fully represent the target device. Thus, two kinds of tests are recommended:

1. Unit tests which test program logic on a Linux machine, isolated through mocks.
2. System/Integration tests which test the interaction of components and the whole system. They run on the target, where irrelevant components and code may as well be emulated via mocks.

This documentation is about the first kind of tests. Refer to *target based unit testing* for more information on target tests (the second kind of tests).

4.30.2 IDF Unit Tests on Linux Host

The current focus of the Linux host tests is on creating isolated unit tests of components, while mocking the component's dependencies with CMock.

A complete implementation of IDF to run on Linux does not exist currently.

There are currently two examples for running IDF-built code on Linux host:

- An example hello-world application
- A unit test for NVS .

Inside the component which should be tested, there is a separate directory `host_test`, besides the "traditional" `test` directory or the `test_apps` directory. It has one or more subdirectories:

```
- host_test/

              - fixtures/
                  contains test fixtures (structs/functions to do test case set-up␣
↪and tear-down).
                  If there are no fixtures, this can be ommitted.
              - <test_name>/
                  IDF applications which run the tests
              - <test_name2>/
                  Further tests are possible.
```

The IDF applications inside `host_test` set the mocking configuration as described in the *IDF unit test documentation*.

The NVS page unit test provides some illustration of how to control the mocks.

Requirements

- Installed ESP-IDF including all ESP-IDF requirements
- System package requirements (`libbsd`, `libbsd-dev`)
- A recent enough Linux or macOS version and GCC compiler
- All components the application depends on must be either supported on the Linux target (Linux/POSIX simulator) or mock-able

An application that runs on the Linux target has to set the `COMPONENTS` variable to `main` in the CMakeLists.txt of the application's root directory:

```
set(COMPONENTS main)
```

This prevents the automatic inclusion of all components from ESP-IDF to the build process which is otherwise done for convenience.

The host tests have been tested on Ubuntu 20.04 with `GCC` version 9 and 10.

4.31 Wi-Fi Driver

4.31.1 ESP32 Wi-Fi Feature List

The following features are supported:

- 4 virtual Wi-Fi interfaces, which are STA, AP, Sniffer and reserved.
- Station-only mode, AP-only mode, station/AP-coexistence mode
- IEEE 802.11b, IEEE 802.11g, IEEE 802.11n, and APIs to configure the protocol mode
- WPA/WPA2/WPA3/WPA2-Enterprise/WPA3-Enterprise/WAPI/WPS and DPP
- AMSDU, AMPDU, HT40, QoS, and other key features
- Modem-sleep
- The Espressif-specific ESP-NOW protocol and Long Range mode, which supports up to **1 km** of data traffic
- Up to 20 MBit/s TCP throughput and 30 MBit/s UDP throughput over the air
- Sniffer
- Both fast scan and all-channel scan
- Multiple antennas
- Channel state information

4.31.2 How To Write a Wi-Fi Application

Preparation

Generally, the most effective way to begin your own Wi-Fi application is to select an example which is similar to your own application, and port the useful part into your project. It is not a MUST, but it is strongly recommended that you take some time to read this article first, especially if you want to program a robust Wi-Fi application.

This article is supplementary to the Wi-Fi APIs/Examples. It describes the principles of using the Wi-Fi APIs, the limitations of the current Wi-Fi API implementation, and the most common pitfalls in using Wi-Fi. This article also reveals some design details of the Wi-Fi driver. We recommend you to select an example .

Setting Wi-Fi Compile-time Options

Refer to *Wi-Fi Menuconfig*.

Init Wi-Fi

Refer to *ESP32 Wi-Fi station General Scenario* and *ESP32 Wi-Fi AP General Scenario*.

Start/Connect Wi-Fi

Refer to *ESP32 Wi-Fi station General Scenario* and *ESP32 Wi-Fi AP General Scenario*.

Event-Handling

Generally, it is easy to write code in "sunny-day" scenarios, such as *WIFI_EVENT_STA_START* and *WIFI_EVENT_STA_CONNECTED*. The hard part is to write routines in "rainy-day" scenarios, such as *WIFI_EVENT_STA_DISCONNECTED*. Good handling of "rainy-day" scenarios is fundamental to robust Wi-Fi applications. Refer to *ESP32 Wi-Fi Event Description*, *ESP32 Wi-Fi station General Scenario*, and *ESP32 Wi-Fi AP General Scenario*. See also *an overview of event handling in ESP-IDF*.

Write Error-Recovery Routines Correctly at All Times

Just like the handling of "rainy-day" scenarios, a good error-recovery routine is also fundamental to robust Wi-Fi applications. Refer to *ESP32 Wi-Fi API Error Code*.

4.31.3 ESP32 Wi-Fi API Error Code

All of the ESP32 Wi-Fi APIs have well-defined return values, namely, the error code. The error code can be categorized into:

- No errors, e.g., ESP_OK means that the API returns successfully.
- Recoverable errors, such as ESP_ERR_NO_MEM.
- Non-recoverable, non-critical errors.
- Non-recoverable, critical errors.

Whether the error is critical or not depends on the API and the application scenario, and it is defined by the API user.

The primary principle to write a robust application with Wi-Fi API is to always check the error code and write the error-handling code. Generally, the error-handling code can be used:

- For recoverable errors, in which case you can write a recoverable-error code. For example, when `esp_wifi_start()` returns ESP_ERR_NO_MEM, the recoverable-error code vTaskDelay can be called in order to get a microseconds' delay for another try.
- For non-recoverable, yet non-critical errors, in which case printing the error code is a good method for error handling.
- For non-recoverable and also critical errors, in which case "assert" may be a good method for error handling. For example, if `esp_wifi_set_mode()` returns ESP_ERR_WIFI_NOT_INIT, it means that the Wi-Fi driver is not initialized by `esp_wifi_init()` successfully. You can detect this kind of error very quickly in the application development phase.

In esp_err.h, ESP_ERROR_CHECK checks the return values. It is a rather commonplace error-handling code and can be used as the default error-handling code in the application development phase. However, it is strongly recommended that API users write their own error-handling code.

4.31.4 ESP32 Wi-Fi API Parameter Initialization

When initializing struct parameters for the API, one of two approaches should be followed:

- Explicitly set all fields of the parameter.
- Use get API to get current configuration first, then set application specific fields.

Initializing or getting the entire structure is very important, because most of the time the value 0 indicates that the default value is used. More fields may be added to the struct in the future and initializing these to zero ensures the application will still work correctly after ESP-IDF is updated to a new release.

4.31.5 ESP32 Wi-Fi Programming Model

The ESP32 Wi-Fi programming model is depicted as follows:

The Wi-Fi driver can be considered a black box that knows nothing about high-layer code, such as the TCP/IP stack, application task, and event task. The application task (code) generally calls *Wi-Fi driver APIs* to initialize Wi-Fi and handles Wi-Fi events when necessary. Wi-Fi driver receives API calls, handles them, and posts events to the application.

Wi-Fi event handling is based on the *esp_event library*. Events are sent by the Wi-Fi driver to the *default event loop*. Application may handle these events in callbacks registered using `esp_event_handler_register()`. Wi-Fi events are also handled by *esp_netif component* to provide a set of default behaviors. For example, when Wi-Fi station connects to an AP, esp_netif will automatically start the DHCP client by default.

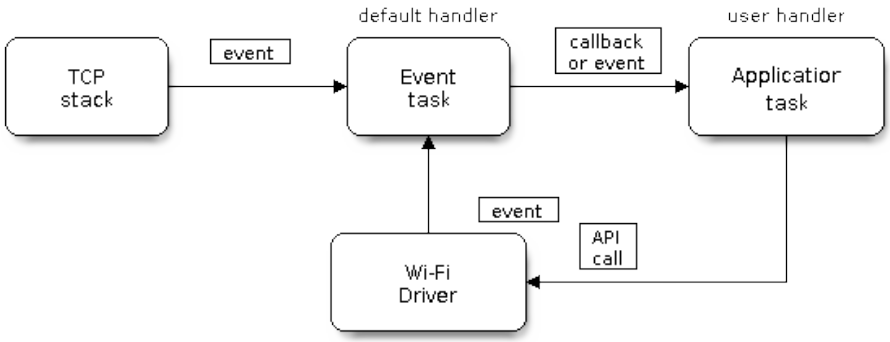


Fig. 55: Wi-Fi Programming Model

4.31.6 ESP32 Wi-Fi Event Description

WIFI_EVENT_WIFI_READY

The Wi-Fi driver will never generate this event, which, as a result, can be ignored by the application event callback. This event may be removed in future releases.

WIFI_EVENT_SCAN_DONE

The scan-done event is triggered by `esp_wifi_scan_start()` and will arise in the following scenarios:

- The scan is completed, e.g., the target AP is found successfully, or all channels have been scanned.
- The scan is stopped by `esp_wifi_scan_stop()`.
- The `esp_wifi_scan_start()` is called before the scan is completed. A new scan will override the current scan and a scan-done event will be generated.

The scan-done event will not arise in the following scenarios:

- It is a blocked scan.
- The scan is caused by `esp_wifi_connect()`.

Upon receiving this event, the event task does nothing. The application event callback needs to call `esp_wifi_scan_get_ap_num()` and `esp_wifi_scan_get_ap_records()` to fetch the scanned AP list and trigger the Wi-Fi driver to free the internal memory which is allocated during the scan **(do not forget to do this!)**. Refer to *ESP32 Wi-Fi Scan* for a more detailed description.

WIFI_EVENT_STA_START

If `esp_wifi_start()` returns ESP_OK and the current Wi-Fi mode is station or station/AP, then this event will arise. Upon receiving this event, the event task will initialize the LwIP network interface (netif). Generally, the application event callback needs to call `esp_wifi_connect()` to connect to the configured AP.

WIFI_EVENT_STA_STOP

If `esp_wifi_stop()` returns ESP_OK and the current Wi-Fi mode is station or station/AP, then this event will arise. Upon receiving this event, the event task will release the station's IP address, stop the DHCP client, remove TCP/UDP-related connections, and clear the LwIP station netif, etc. The application event callback generally does not need to do anything.

WIFI_EVENT_STA_CONNECTED

If `esp_wifi_connect()` returns ESP_OK and the station successfully connects to the target AP, the connection event will arise. Upon receiving this event, the event task starts the DHCP client and begins the DHCP process of getting the IP address. Then, the Wi-Fi driver is ready for sending and receiving data. This moment is good for beginning the application work, provided that the application does not depend on LwIP, namely the IP address. However, if the application is LwIP-based, then you need to wait until the *got ip* event comes in.

WIFI_EVENT_STA_DISCONNECTED

This event can be generated in the following scenarios:

- When `esp_wifi_disconnect()` or `esp_wifi_stop()` is called and the station is already connected to the AP.
- When `esp_wifi_connect()` is called, but the Wi-Fi driver fails to set up a connection with the AP due to certain reasons, e.g., the scan fails to find the target AP or the authentication times out. If there are more than one AP with the same SSID, the disconnected event will be raised after the station fails to connect all of the found APs.
- When the Wi-Fi connection is disrupted because of specific reasons, e.g., the station continuously loses N beacons, the AP kicks off the station, or the AP' s authentication mode is changed.

Upon receiving this event, the default behaviors of the event task are:

- Shutting down the station' s LwIP netif.
- Notifying the LwIP task to clear the UDP/TCP connections which cause the wrong status to all sockets. For socket-based applications, the application callback can choose to close all sockets and re-create them, if necessary, upon receiving this event.

The most common event handle code for this event in application is to call `esp_wifi_connect()` to reconnect the Wi-Fi. However, if the event is raised because `esp_wifi_disconnect()` is called, the application should not call `esp_wifi_connect()` to reconnect. It is the application' s responsibility to distinguish whether the event is caused by `esp_wifi_disconnect()` or other reasons. Sometimes a better reconnection strategy is required. Refer to *Wi-Fi Reconnect* and *Scan When Wi-Fi Is Connecting*.

Another thing that deserves attention is that the default behavior of LwIP is to abort all TCP socket connections on receiving the disconnect. In mose cases, it is not a problem. However, for some special applications, this may not be what they want. Consider the following scenarios:

- The application creates a TCP connection to maintain the application-level keep-alive data that is sent out every 60 seconds.
- Due to certain reasons, the Wi-Fi connection is cut off, and the *WIFI_EVENT_STA_DISCONNECTED* is raised. According to the current implementation, all TCP connections will be removed and the keep-alive socket will be in a wrong status. However, since the application designer believes that the network layer should **ignore** this error at the Wi-Fi layer, the application does not close the socket.
- Five seconds later, the Wi-Fi connection is restored because `esp_wifi_connect()` is called in the application event callback function. **Moreover, the station connects to the same AP and gets the same IPV4 address as before**.
- Sixty seconds later, when the application sends out data with the keep-alive socket, the socket returns an error and the application closes the socket and re-creates it when necessary.

In above scenarios, ideally, the application sockets and the network layer should not be affected, since the Wi-Fi connection only fails temporarily and recovers very quickly. The application can enable "Keep TCP connections when IP changed" via LwIP menuconfig.

IP_EVENT_STA_GOT_IP

This event arises when the DHCP client successfully gets the IPV4 address from the DHCP server, or when the IPV4 address is changed. The event means that everything is ready and the application can begin its tasks (e.g., creating sockets).

The IPV4 may be changed because of the following reasons:

- The DHCP client fails to renew/rebind the IPV4 address, and the station's IPV4 is reset to 0.
- The DHCP client rebinds to a different address.
- The static-configured IPV4 address is changed.

Whether the IPV4 address is changed or not is indicated by the field `ip_change` of `ip_event_got_ip_t`.

The socket is based on the IPV4 address, which means that, if the IPV4 changes, all sockets relating to this IPV4 will become abnormal. Upon receiving this event, the application needs to close all sockets and recreate the application when the IPV4 changes to a valid one.

IP_EVENT_GOT_IP6

This event arises when the IPV6 SLAAC support auto-configures an address for the ESP32, or when this address changes. The event means that everything is ready and the application can begin its tasks, e.g., creating sockets.

IP_EVENT_STA_LOST_IP

This event arises when the IPV4 address becomes invalid.

IP_EVENT_STA_LOST_IP does not arise immediately after the Wi-Fi disconnects. Instead, it starts an IPV4 address lost timer. If the IPV4 address is got before ip lost timer expires, IP_EVENT_STA_LOST_IP does not happen. Otherwise, the event arises when the IPV4 address lost timer expires.

Generally, the application can ignore this event, because it is just a debug event to inform that the IPV4 address is lost.

WIFI_EVENT_AP_START

Similar to *WIFI_EVENT_STA_START*.

WIFI_EVENT_AP_STOP

Similar to *WIFI_EVENT_STA_STOP*.

WIFI_EVENT_AP_STACONNECTED

Every time a station is connected to ESP32 AP, the *WIFI_EVENT_AP_STACONNECTED* will arise. Upon receiving this event, the event task will do nothing, and the application callback can also ignore it. However, you may want to do something, for example, to get the info of the connected STA.

WIFI_EVENT_AP_STADISCONNECTED

This event can happen in the following scenarios:

- The application calls *esp_wifi_disconnect()*, or *esp_wifi_deauth_sta()*, to manually disconnect the station.
- The Wi-Fi driver kicks off the station, e.g., because the AP has not received any packets in the past five minutes. The time can be modified by *esp_wifi_set_inactive_time()*.
- The station kicks off the AP.

When this event happens, the event task will do nothing, but the application event callback needs to do something, e.g., close the socket which is related to this station.

WIFI_EVENT_AP_PROBEREQRECVED

This event is disabled by default. The application can enable it via API `esp_wifi_set_event_mask()`. When this event is enabled, it will be raised each time the AP receives a probe request.

WIFI_EVENT_STA_BEACON_TIMEOUT

If the station does not receive the beacon of the connected AP within the inactive time, the beacon timeout happens, the *WIFI_EVENT_STA_BEACON_TIMEOUT* will arise. The application can set inactive time via API `esp_wifi_set_inactive_time()`.

WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START

The *WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START* will arise at the start of connectionless module *Interval*. See *connectionless module power save*.

4.31.7 ESP32 Wi-Fi Station General Scenario

Below is a "big scenario" which describes some small scenarios in station mode:

1. Wi-Fi/LwIP Init Phase

- s1.1: The main task calls `esp_netif_init()` to create an LwIP core task and initialize LwIP-related work.
- s1.2: The main task calls `esp_event_loop_create()` to create a system Event task and initialize an application event's callback function. In the scenario above, the application event's callback function does nothing but relaying the event to the application task.
- s1.3: The main task calls `esp_netif_create_default_wifi_ap()` or `esp_netif_create_default_wifi_sta()` to create default network interface instance binding station or AP with TCP/IP stack.
- s1.4: The main task calls `esp_wifi_init()` to create the Wi-Fi driver task and initialize the Wi-Fi driver.
- s1.5: The main task calls OS API to create the application task.

Step 1.1 ~ 1.5 is a recommended sequence that initializes a Wi-Fi-/LwIP-based application. However, it is **NOT** a must-follow sequence, which means that you can create the application task in step 1.1 and put all other initializations in the application task. Moreover, you may not want to create the application task in the initialization phase if the application task depends on the sockets. Rather, you can defer the task creation until the IP is obtained.

2. Wi-Fi Configuration Phase

Once the Wi-Fi driver is initialized, you can start configuring the Wi-Fi driver. In this scenario, the mode is station, so you may need to call `esp_wifi_set_mode()` (WIFI_MODE_STA) to configure the Wi-Fi mode as station. You can call other *esp_wifi_set_xxx* APIs to configure more settings, such as the protocol mode, the country code, and the bandwidth. Refer to *ESP32 Wi-Fi Configuration*.

Generally, the Wi-Fi driver should be configured before the Wi-Fi connection is set up. But this is **NOT** mandatory, which means that you can configure the Wi-Fi connection anytime, provided that the Wi-Fi driver is initialized successfully. However, if the configuration does not need to change after the Wi-Fi connection is set up, you should configure the Wi-Fi driver at this stage, because the configuration APIs (such as `esp_wifi_set_protocol()`) will cause the Wi-Fi to reconnect, which may not be desirable.

If the Wi-Fi NVS flash is enabled by menuconfig, all Wi-Fi configuration in this phase, or later phases, will be stored into flash. When the board powers on/reboots, you do not need to configure the Wi-Fi driver from scratch. You only need to call esp_wifi_get_xxx APIs to fetch the configuration stored in flash previously. You can also configure the Wi-Fi driver if the previous configuration is not what you want.

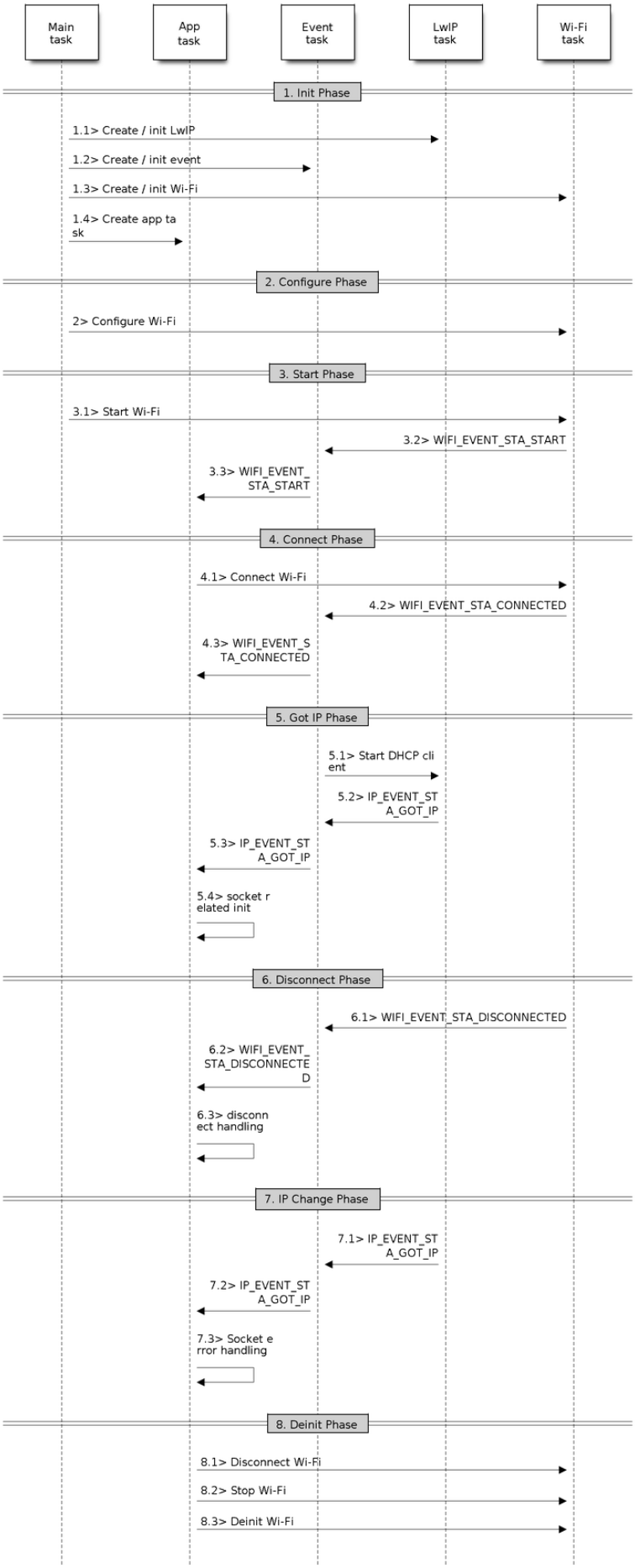


Fig. 56: Sample Wi-Fi Event Scenarios in Station Mode

3. Wi-Fi Start Phase

- s3.1: Call `esp_wifi_start()` to start the Wi-Fi driver.
- s3.2: The Wi-Fi driver posts *WIFI_EVENT_STA_START* to the event task; then, the event task will do some common things and will call the application event callback function.
- s3.3: The application event callback function relays the *WIFI_EVENT_STA_START* to the application task. We recommend that you call `esp_wifi_connect()`. However, you can also call `esp_wifi_connect()` in other phrases after the *WIFI_EVENT_STA_START* arises.

4. Wi-Fi Connect Phase

- s4.1: Once `esp_wifi_connect()` is called, the Wi-Fi driver will start the internal scan/connection process.
- s4.2: If the internal scan/connection process is successful, the *WIFI_EVENT_STA_CONNECTED* will be generated. In the event task, it starts the DHCP client, which will finally trigger the DHCP process.
- s4.3: In the above-mentioned scenario, the application event callback will relay the event to the application task. Generally, the application needs to do nothing, and you can do whatever you want, e.g., print a log.

In step 4.2, the Wi-Fi connection may fail because, for example, the password is wrong, or the AP is not found. In a case like this, *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason for such a failure will be provided. For handling events that disrupt Wi-Fi connection, please refer to phase 6.

5. Wi-Fi 'Got IP' Phase

- s5.1: Once the DHCP client is initialized in step 4.2, the *got IP* phase will begin.
- s5.2: If the IP address is successfully received from the DHCP server, then *IP_EVENT_STA_GOT_IP* will arise and the event task will perform common handling.
- s5.3: In the application event callback, *IP_EVENT_STA_GOT_IP* is relayed to the application task. For LwIP-based applications, this event is very special and means that everything is ready for the application to begin its tasks, e.g., creating the TCP/UDP socket. A very common mistake is to initialize the socket before *IP_EVENT_STA_GOT_IP* is received. **DO NOT start the socket-related work before the IP is received.**

6. Wi-Fi Disconnect Phase

- s6.1: When the Wi-Fi connection is disrupted, e.g., the AP is powered off or the RSSI is poor, *WIFI_EVENT_STA_DISCONNECTED* will arise. This event may also arise in phase 3. Here, the event task will notify the LwIP task to clear/remove all UDP/TCP connections. Then, all application sockets will be in a wrong status. In other words, no socket can work properly when this event happens.
- s6.2: In the scenario described above, the application event callback function relays *WIFI_EVENT_STA_DISCONNECTED* to the application task. The recommended actions are: 1) call `esp_wifi_connect()` to reconnect the Wi-Fi, 2) close all sockets, and 3) re-create them if necessary. For details, please refer to *WIFI_EVENT_STA_DISCONNECTED*.

7. Wi-Fi IP Change Phase

- s7.1: If the IP address is changed, the *IP_EVENT_STA_GOT_IP* will arise with "ip_change" set to true.
- s7.2: **This event is important to the application. When it occurs, the timing is good for closing all created sockets and recreating them.**

8. Wi-Fi Deinit Phase

- s8.1: Call `esp_wifi_disconnect()` to disconnect the Wi-Fi connectivity.
- s8.2: Call `esp_wifi_stop()` to stop the Wi-Fi driver.
- s8.3: Call `esp_wifi_deinit()` to unload the Wi-Fi driver.

4.31.8 ESP32 Wi-Fi AP General Scenario

Below is a "big scenario" which describes some small scenarios in AP mode:

4.31.9 ESP32 Wi-Fi Scan

Currently, the `esp_wifi_scan_start()` API is supported only in station or station/AP mode.

Scan Type

Mode	Description
Active Scan	Scan by sending a probe request. The default scan is an active scan.
Passive Scan	No probe request is sent out. Just switch to the specific channel and wait for a beacon. Application can enable it via the scan_type field of wifi_scan_config_t.
Foreground Scan	This scan is applicable when there is no Wi-Fi connection in station mode. Foreground or background scanning is controlled by the Wi-Fi driver and cannot be configured by the application.
Background Scan	This scan is applicable when there is a Wi-Fi connection in station mode or in station/AP mode. Whether it is a foreground scan or background scan depends on the Wi-Fi driver and cannot be configured by the application.
All-Channel Scan	It scans all of the channels. If the channel field of wifi_scan_config_t is set to 0, it is an all-channel scan.
Specific Channel Scan	It scans specific channels only. If the channel field of wifi_scan_config_t set to 1-14, it is a specific-channel scan.

The scan modes in above table can be combined arbitrarily, so there are in total 8 different scans:

- All-Channel Background Active Scan
- All-Channel Background Passive Scan
- All-Channel Foreground Active Scan
- All-Channel Foreground Passive Scan
- Specific-Channel Background Active Scan
- Specific-Channel Background Passive Scan
- Specific-Channel Foreground Active Scan
- Specific-Channel Foreground Passive Scan

Scan Configuration

The scan type and other per-scan attributes are configured by `esp_wifi_scan_start()`. The table below provides a detailed description of wifi_scan_config_t.

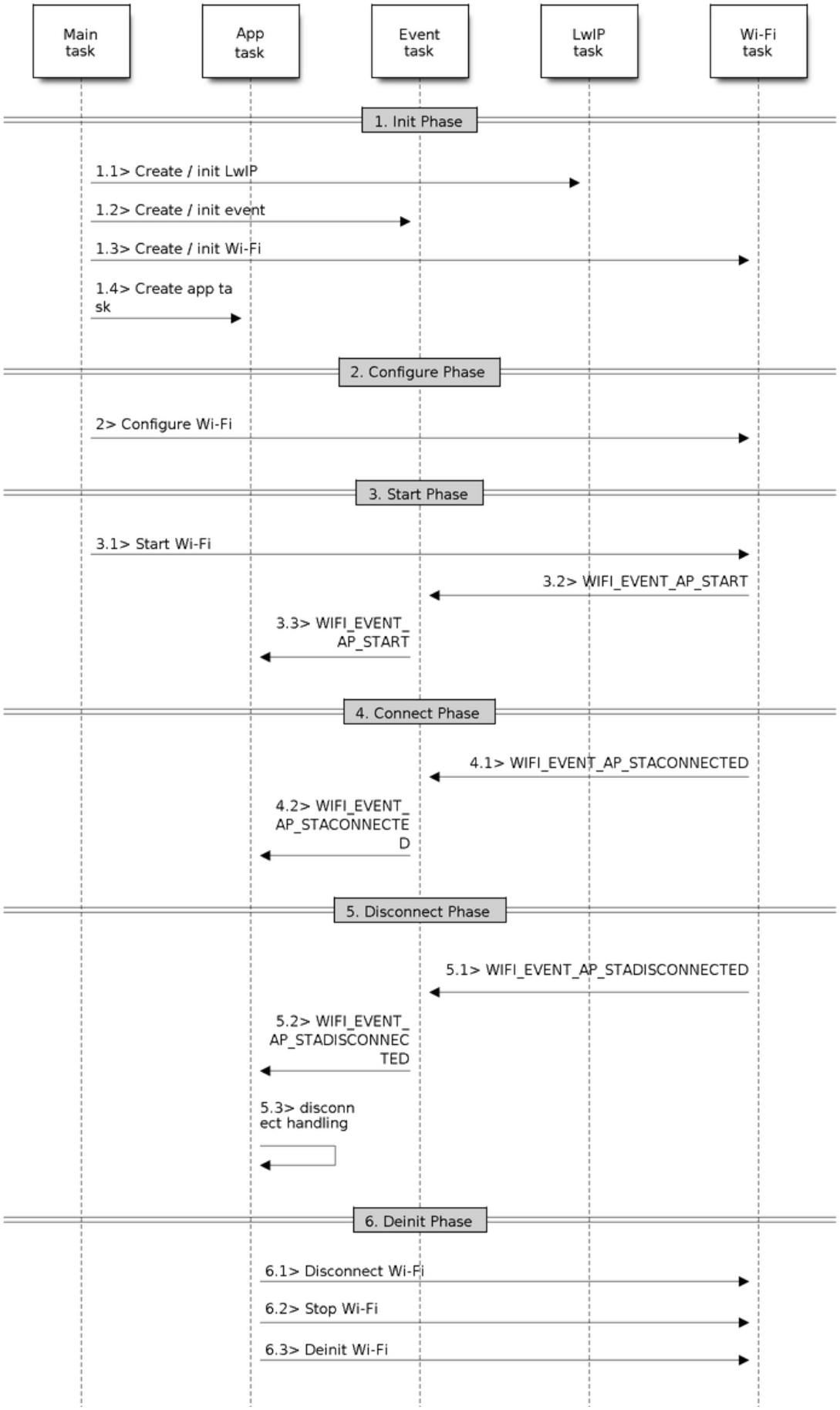


Fig. 57: Sample Wi-Fi Event Scenarios in AP Mode

Field	Description
ssid	If the SSID is not NULL, it is only the AP with the same SSID that can be scanned.
bssid	If the BSSID is not NULL, it is only the AP with the same BSSID that can be scanned.
channel	If "channel" is 0, there will be an all-channel scan; otherwise, there will be a specific-channel scan.
show_hidden	If "show_hidden" is 0, the scan ignores the AP with a hidden SSID; otherwise, the scan considers the hidden AP a normal one.
scan_type	If "scan_type" is WIFI_SCAN_TYPE_ACTIVE, the scan is "active"; otherwise, it is a "passive" one.
scan_time	This field is used to control how long the scan dwells on each channel. For passive scans, scan_time.passive designates the dwell time for each channel. For active scans, dwell times for each channel are listed in the table below. Here, min is short for scan time.active.min and max is short for scan_time.active.max. • min=0, max=0: scan dwells on each channel for 120 ms. • min>0, max=0: scan dwells on each channel for 120 ms. • min=0, max>0: scan dwells on each channel for `max` ms. • min>0, max>0: the minimum time the scan dwells on each channel is `min` ms. If no AP is found during this time frame, the scan switches to the next channel. Otherwise, the scan dwells on the channel for `max` ms. If you want to improve the performance of the the scan, you can try to modify these two parameters.

There are also some global scan attributes which are configured by API `esp_wifi_set_config()`, refer to *Station Basic Configuration*

Scan All APs on All Channels (Foreground)

Scenario:

The scenario above describes an all-channel, foreground scan. The foreground scan can only occur in station mode where the station does not connect to any AP. Whether it is a foreground or background scan is totally determined by the Wi-Fi driver, and cannot be configured by the application.

Detailed scenario description:

Scan Configuration Phase

- s1.1: Call `esp_wifi_set_country()` to set the country info if the default country info is not what you want. Refer to *Wi-Fi Country Code*.
- s1.2: Call `esp_wifi_scan_start()` to configure the scan. To do so, you can refer to *Scan Configuration*. Since this is an all-channel scan, just set the SSID/BSSID/channel to 0.

Wi-Fi Driver's Internal Scan Phase

- s2.1: The Wi-Fi driver switches to channel 1. In this case, the scan type is WIFI_SCAN_TYPE_ACTIVE, and a probe request is broadcasted. Otherwise, the Wi-Fi will wait for a beacon from the APs. The Wi-Fi driver will stay in channel 1 for some time. The dwell time is configured in min/max time, with the default value being 120 ms.
- s2.2: The Wi-Fi driver switches to channel 2 and performs the same operation as in step 2.1.
- s2.3: The Wi-Fi driver scans the last channel N, where N is determined by the country code which is configured in step 1.1.

Scan-Done Event Handling Phase

- s3.1: When all channels are scanned, *WIFI_EVENT_SCAN_DONE* will arise.

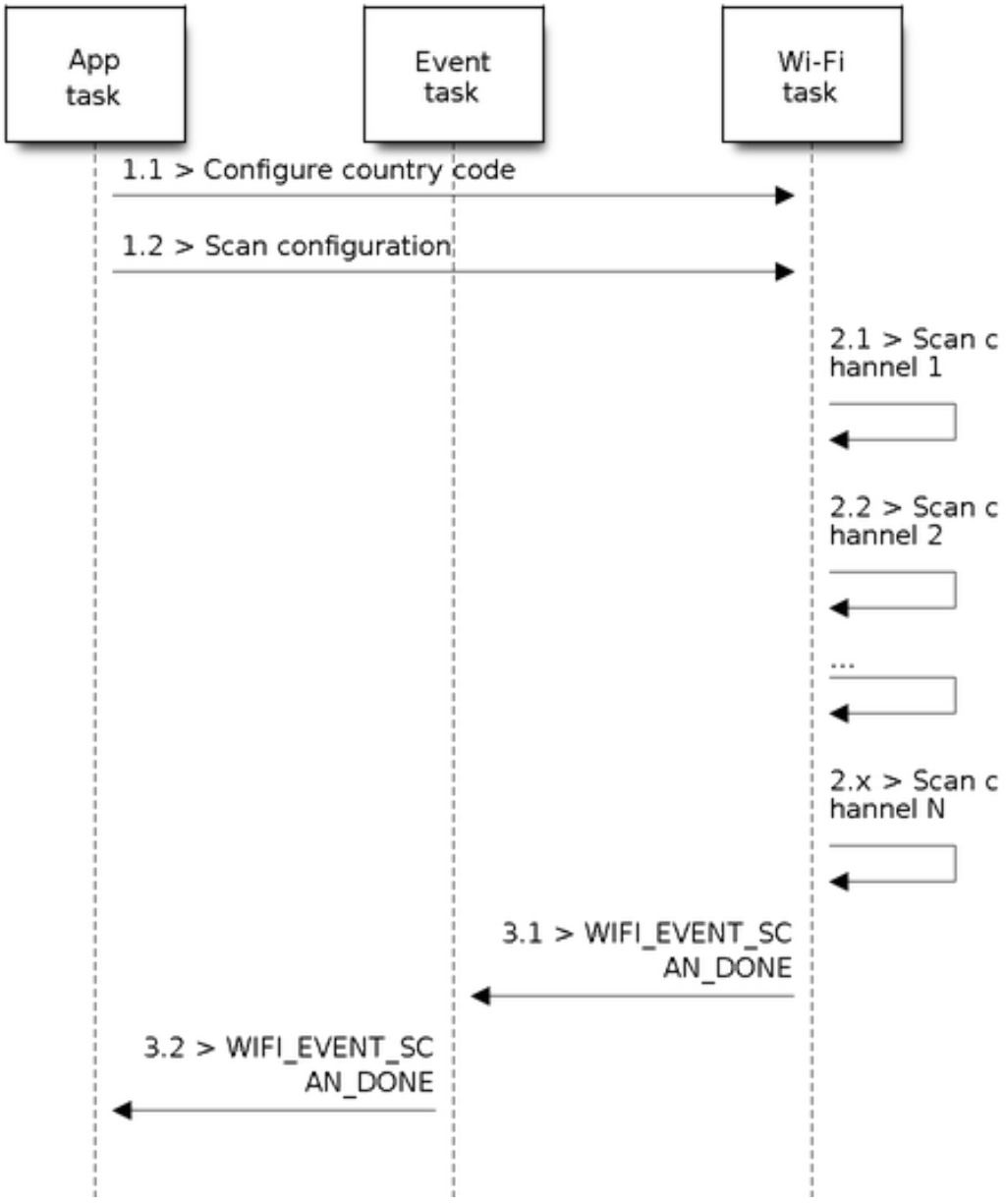


Fig. 58: Foreground Scan of all Wi-Fi Channels

- s3.2: The application's event callback function notifies the application task that *WIFI_EVENT_SCAN_DONE* is received. `esp_wifi_scan_get_ap_num()` is called to get the number of APs that have been found in this scan. Then, it allocates enough entries and calls `esp_wifi_scan_get_ap_records()` to get the AP records. Please note that the AP records in the Wi-Fi driver will be freed once `esp_wifi_scan_get_ap_records()` is called. Do not call `esp_wifi_scan_get_ap_records()` twice for a single scan-done event. If `esp_wifi_scan_get_ap_records()` is not called when the scan-done event occurs, the AP records allocated by the Wi-Fi driver will not be freed. So, make sure you call `esp_wifi_scan_get_ap_records()`, yet only once.

Scan All APs on All Channels (Background)

Scenario:

The scenario above is an all-channel background scan. Compared to *Scan All APs on All Channels (Foreground)* , the difference in the all-channel background scan is that the Wi-Fi driver will scan the back-to-home channel for 30 ms before it switches to the next channel to give the Wi-Fi connection a chance to transmit/receive data.

Scan for Specific AP on All Channels

Scenario:

This scan is similar to *Scan All APs on All Channels (Foreground)*. The differences are:

- s1.1: In step 1.2, the target AP will be configured to SSID/BSSID.
- s2.1 ~ s2.N: Each time the Wi-Fi driver scans an AP, it will check whether it is a target AP or not. If the scan is WIFI_FAST_SCAN scan and the target AP is found, then the scan-done event will arise and scanning will end; otherwise, the scan will continue. Please note that the first scanned channel may not be channel 1, because the Wi-Fi driver optimizes the scanning sequence.

It is a possible situation that there are multiple APs that match the target AP info, e.g., two APs with the SSID of "ap" are scanned. In this case, if the scan is WIFI_FAST_SCAN, then only the first scanned "ap" will be found. If the scan is WIFI_ALL_CHANNEL_SCAN, both "ap" will be found and the station will connect the "ap" according to the configured strategy. Refer to *Station Basic Configuration*.

You can scan a specific AP, or all of them, in any given channel. These two scenarios are very similar.

Scan in Wi-Fi Connect

When `esp_wifi_connect()` is called, the Wi-Fi driver will try to scan the configured AP first. The scan in "Wi-Fi Connect" is the same as *Scan for Specific AP On All Channels*, except that no scan-done event will be generated when the scan is completed. If the target AP is found, the Wi-Fi driver will start the Wi-Fi connection; otherwise, *WIFI_EVENT_STA_DISCONNECTED* will be generated. Refer to *Scan for Specific AP On All Channels*.

Scan in Blocked Mode

If the block parameter of `esp_wifi_scan_start()` is true, then the scan is a blocked one, and the application task will be blocked until the scan is done. The blocked scan is similar to an unblocked one, except that no scan-done event will arise when the blocked scan is completed.

Parallel Scan

Two application tasks may call `esp_wifi_scan_start()` at the same time, or the same application task calls `esp_wifi_scan_start()` before it gets a scan-done event. Both scenarios can happen. **However, the Wi-Fi driver does not support multiple concurrent scans adequately. As a result, concurrent scans should be avoided.** Support for concurrent scan will be enhanced in future releases, as the ESP32's Wi-Fi functionality improves continuously.

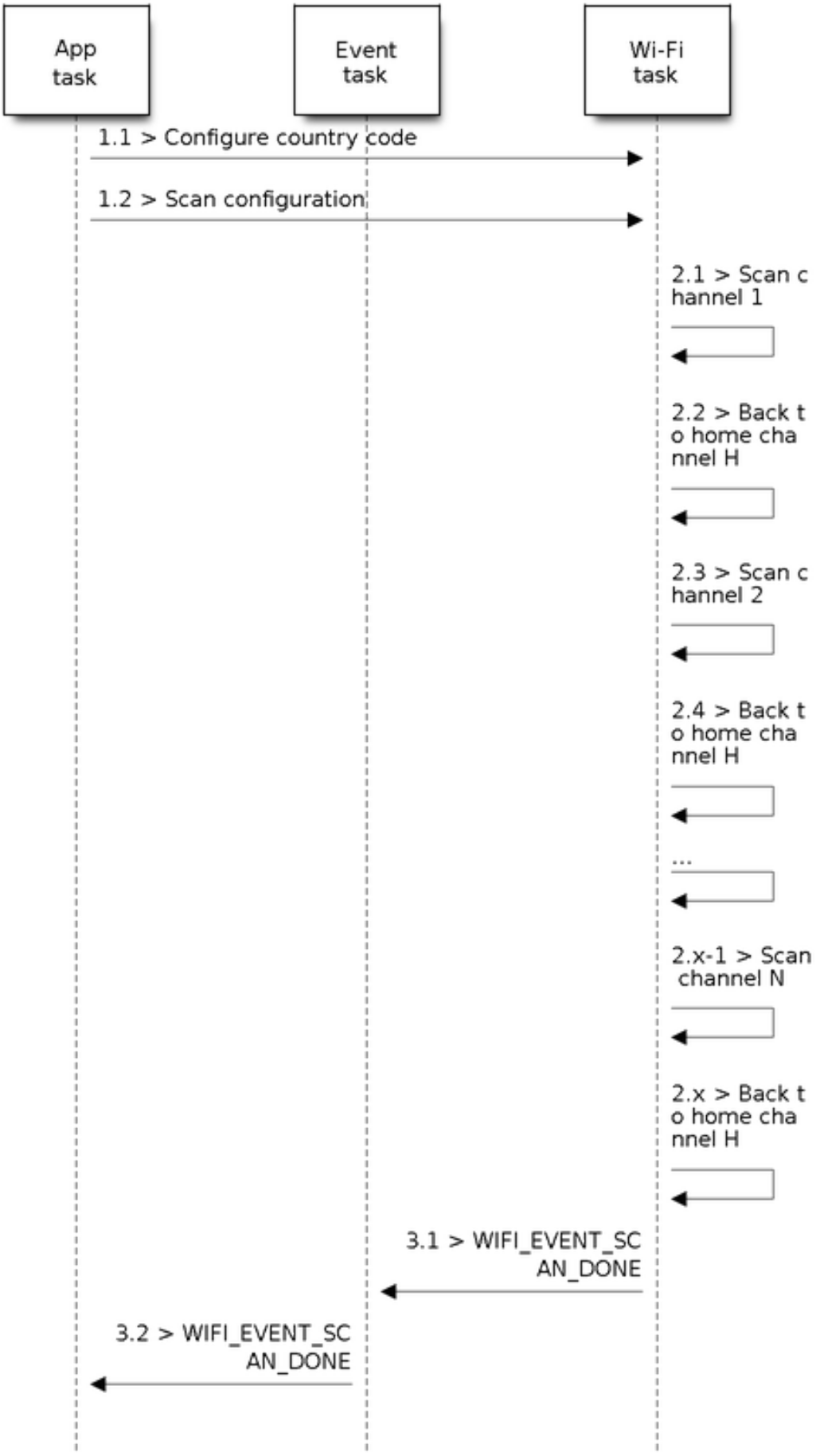


Fig. 59: Background Scan of all Wi-Fi Channels

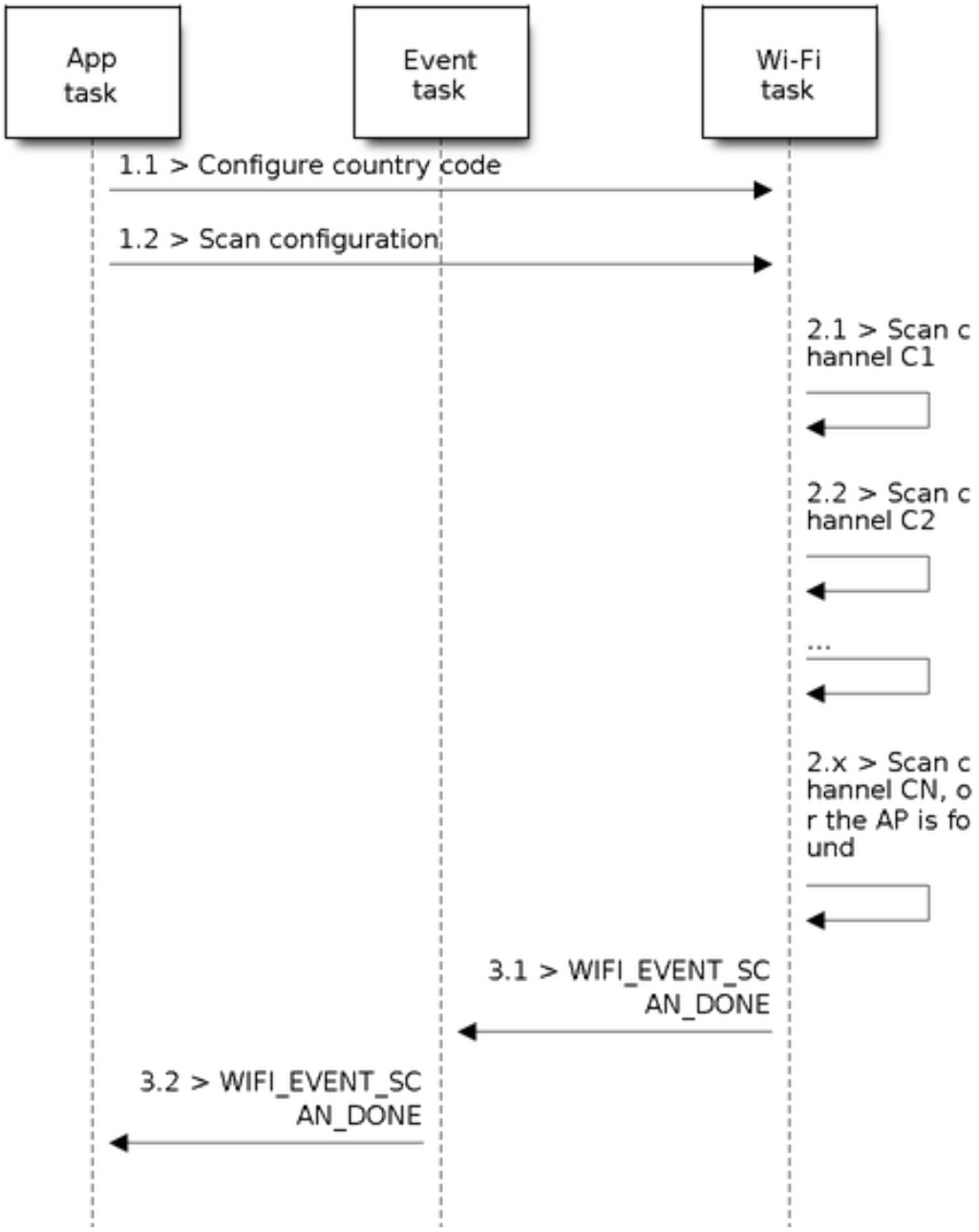


Fig. 60: Scan of specific Wi-Fi Channels

Scan When Wi-Fi Is Connecting

The `esp_wifi_scan_start()` fails immediately if the Wi-Fi is connecting, because the connecting has higher priority than the scan. If scan fails because of connecting, the recommended strategy is to delay for some time and retry scan again. The scan will succeed once the connecting is completed.

However, the retry/delay strategy may not work all the time. Considering the following scenarios:

- The station is connecting a non-existing AP or it connects the existing AP with a wrong password, it always raises the event *WIFI_EVENT_STA_DISCONNECTED*.
- The application calls `esp_wifi_connect()` to reconnect on receiving the disconnect event.
- Another application task, e.g., the console task, calls `esp_wifi_scan_start()` to do scan, the scan always fails immediately because the station keeps connecting.
- When scan fails, the application simply delays for some time and retries the scan.

In the above scenarios, the scan will never succeed because the connecting is in process. So if the application supports similar scenario, it needs to implement a better reconnection strategy. For example:

- The application can choose to define a maximum continuous reconnection counter and stop reconnecting once the counter reaches the maximum.
- The application can choose to reconnect immediately in the first N continuous reconnection, then give a delay sometime and reconnect again.

The application can define its own reconnection strategy to avoid the scan starve to death. Refer to <*Wi-Fi Reconnect*>.

4.31.10 ESP32 Wi-Fi Station Connecting Scenario

This scenario depicts the case if only one target AP is found in the scan phase. For scenarios where more than one AP with the same SSID is found, refer to *ESP32 Wi-Fi Station Connecting When Multiple APs Are Found*.

Generally, the application can ignore the connecting process. Below is a brief introduction to the process for those who are really interested.

Scenario:

Scan Phase

- s1.1: The Wi-Fi driver begins scanning in "Wi-Fi Connect". Refer to *Scan in Wi-Fi Connect* for more details.
- s1.2: If the scan fails to find the target AP, *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason code will be WIFI_REASON_NO_AP_FOUND. Refer to *Wi-Fi Reason Code*.

Auth Phase

- s2.1: The authentication request packet is sent and the auth timer is enabled.
- s2.2: If the authentication response packet is not received before the authentication timer times out, *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason code will be WIFI_REASON_AUTH_EXPIRE. Refer to *Wi-Fi Reason Code*.
- s2.3: The auth-response packet is received and the auth-timer is stopped.
- s2.4: The AP rejects authentication in the response and *WIFI_EVENT_STA_DISCONNECTED* arises, while the reason code is WIFI_REASON_AUTH_FAIL or the reasons specified by the AP. Refer to *Wi-Fi Reason Code*.

Association Phase

- s3.1: The association request is sent and the association timer is enabled.
- s3.2: If the association response is not received before the association timer times out, *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason code will be WIFI_REASON_ASSOC_EXPIRE. Refer to *Wi-Fi Reason Code*.

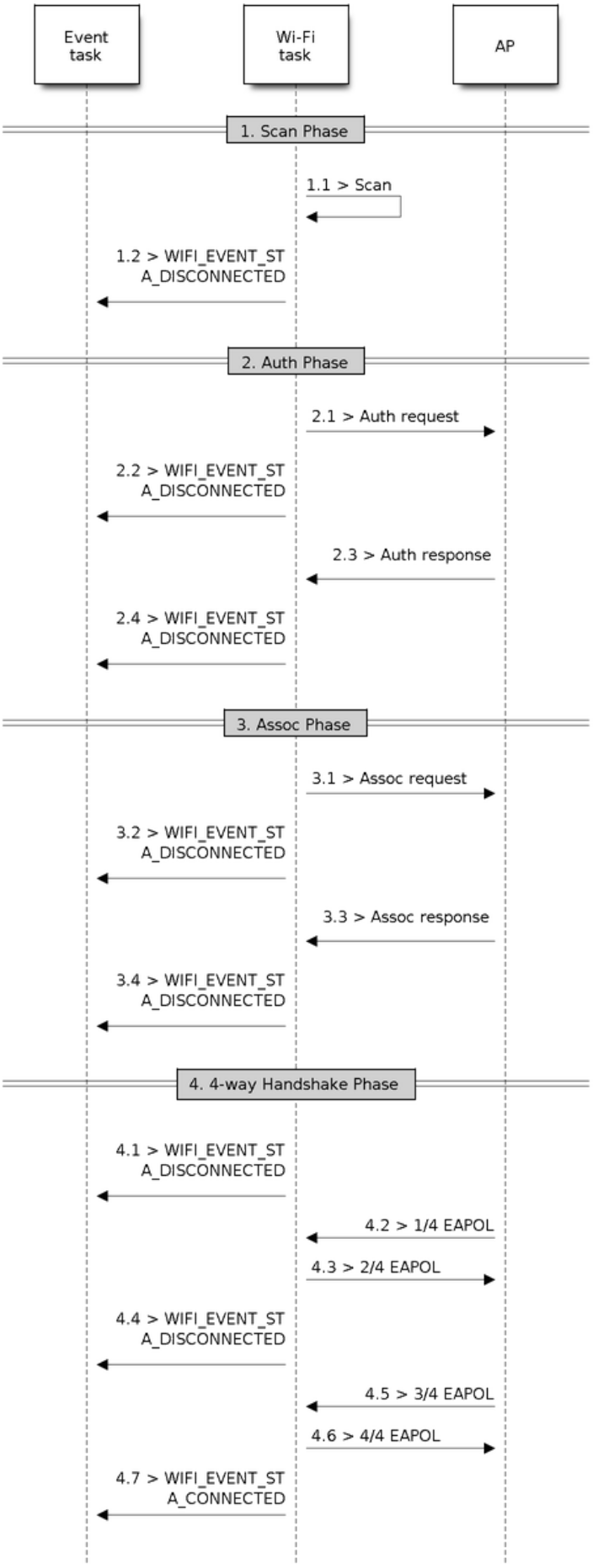


Fig. 61: Wi-Fi Station Connecting Process

- s3.3: The association response is received and the association timer is stopped.
- s3.4: The AP rejects the association in the response and *WIFI_EVENT_STA_DISCONNECTED* arises, while the reason code is the one specified in the association response. Refer to *Wi-Fi Reason Code*.

Four-way Handshake Phase

- s4.1: The handshake timer is enabled, and the 1/4 EAPOL is not received before the handshake timer expires. *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason code will be WIFI_REASON_HANDSHAKE_TIMEOUT. Refer to *Wi-Fi Reason Code*.
- s4.2: The 1/4 EAPOL is received.
- s4.3: The station replies 2/4 EAPOL.
- s4.4: If the 3/4 EAPOL is not received before the handshake timer expires, *WIFI_EVENT_STA_DISCONNECTED* will arise and the reason code will be WIFI_REASON_HANDSHAKE_TIMEOUT. Refer to *Wi-Fi Reason Code*.
- s4.5: The 3/4 EAPOL is received.
- s4.6: The station replies 4/4 EAPOL.
- s4.7: The station raises *WIFI_EVENT_STA_CONNECTED*.

Wi-Fi Reason Code

The table below shows the reason-code defined in ESP32. The first column is the macro name defined in esp_wifi_types.h. The common prefix *WIFI_REASON* is removed, which means that *UNSPECIFIED* actually stands for *WIFI_REASON_UNSPECIFIED* and so on. The second column is the value of the reason. The third column is the standard value to which this reason is mapped in section 9.4.1.7 of IEEE 802.11-2020. (For more information, refer to the standard mentioned above.) The last column describes the reason.

Reason code	Value	Mapped To	Description
UNSPECIFIED	1	1	Generally, it means an internal failure, e.g., the memory runs out, the internal TX fails, or the reason is received from the remote side.
AUTH_EXPIRE	2	2	The previous authentication is no longer valid. For the ESP station, this reason is reported when: • auth is timed out. • the reason is received from the AP. For the ESP AP, this reason is reported when: • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling `esp_wifi_stop()`. • the station is de-authed by calling `esp_wifi_deauth_sta()`.
AUTH_LEAVE	3	3	De-authenticated, because the sending station is leaving (or has left). For the ESP station, this reason is reported when: • it is received from the AP.

continues on next page

Table 30 – continued from previous page

Reason code	Value	Mapped To	Description
ASSOC_EXPIRE	4	4	Disassociated due to inactivity. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling `esp_wifi_stop()`. • the station is de-authed by calling `esp_wifi_deauth_sta()`.
ASSOC_TOOMANY	5	5	Disassociated, because the AP is unable to handle all currently associated STAs at the same time. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the stations associated with the AP reach the maximum number that the AP can support.
NOT_AUTHED	6	6	Class-2 frame received from a non-authenticated STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives a packet with data from a non-authenticated station.
NOT_ASSOCED	7	7	Class-3 frame received from a non-associated STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives a packet with data from a non-associated station.
ASSOC_LEAVE	8	8	Disassociated, because the sending station is leaving (or has left) BSS. For the ESP station, this reason is reported when: • it is received from the AP. • the station is disconnected by `esp_wifi_disconnect()` and other APIs.
ASSOC_NOT_AUTHED	9	9	station requesting (re)association is not authenticated by the responding STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives packets with data from an associated, yet not authenticated, station.
DISASSOC_PWRCAP_BAD	10	10	Disassociated, because the information in the Power Capability element is unacceptable. For the ESP station, this reason is reported when: • it is received from the AP.

continues on next page

Table 30 – continued from previous page

Reason code	Value	Mapped To	Description
DISASSOC_SUPCHAN_BAD	11	11	Disassociated, because the information in the Supported Channels element is unacceptable. For the ESP station, this reason is reported when: • it is received from the AP.
IE_INVALID	13	13	Invalid element, i.e., an element whose content does not meet the specifications of the Standard in frame formats clause. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP parses a wrong WPA or RSN IE.
MIC_FAILURE	14	14	Message integrity code (MIC) failure. For the ESP station, this reason is reported when: • it is received from the AP.
4WAY_HANDSHAKE_TIMEOUT	15	15	Four-way handshake times out. For legacy reasons, in ESP this reason code is replaced with WIFI_REASON_HANDSHAKE_TIMEOUT. For the ESP station, this reason is reported when: • the handshake times out. • it is received from the AP.
GROUP1_KEY_UPDATE_TIMEOUT	16	16	Group-Key Handshake times out. For the ESP station, this reason is reported when: • it is received from the AP.
IE_IN_4WAY_DIFFERS	17	17	The element in the four-way handshake is different from the (Re-)Association Request/Probe and Response/Beacon frame. For the ESP station, this reason is reported when: • it is received from the AP. • the station finds that the four-way handshake IE differs from the IE in the (Re-)Association Request/Probe and Response/Beacon frame.
GROUP1_CIPHER_INVALID	18	18	Invalid group cipher. For the ESP station, this reason is reported when: • it is received from the AP.
PAIRWISE_CIPHER_INVALID	19	19	Invalid pairwise cipher. For the ESP station, this reason is reported when: • it is received from the AP.
AKMP_INVALID	20	20	Invalid AKMP. For the ESP station, this reason is reported when: - it is received from the AP.
UNSUPP_RSN_IE_VERSION	21	21	Unsupported RSNE version. For the ESP station, this reason is reported when: • it is received from the AP.

continues on next page

Table 30 – continued from previous page

Reason code	Value	Mapped To	Description
INVALID_RSN_IE_CAP	22	22	Invalid RSNE capabilities. For the ESP station, this reason is reported when: • it is received from the AP.
802_1X_AUTH_FAILED	23	23	IEEE 802.1X. authentication failed. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • IEEE 802.1X. authentication fails.
CIPHER_SUITE_REJECTED	24	24	Cipher suite rejected due to security policies. For the ESP station, this reason is reported when: • it is received from the AP.
TDLS_PEER_UNREACHABLE	25	25	TDLS direct-link teardown due to TDLS peer STA unreachable via the TDLS direct link.
TDLS_UNSPECIFIED	26	26	TDLS direct-link teardown for unspecified reason.
SSP_REQUESTED_DISASSOC	27	27	Disassociated because session terminated by SSP request.
NO_SSP_ROAMING_AGREEMENT	28	28	Disassociated because of lack of SSP roaming agreement.
BAD_CIPHER_OR_AKM	29	29	Requested service rejected because of SSP cipher suite or AKM requirement.
NOT_AUTHORIZED_THIS_LOCATION	30	30	Requested service not authorized in this location.
SERVICE_CHANGE_PRECLUDES_TS	31	31	TS deleted because QoS AP lacks sufficient bandwidth for this QoS STA due to a change in BSS service characteristics or operational mode (e.g., an HT BSS change from 40 MHz channel to 20 MHz channel).
UNSPECIFIED_QOS	32	32	Disassociated for unspecified, QoS-related reason.
NOT_ENOUGH_BANDWIDTH	33	33	Disassociated because QoS AP lacks sufficient bandwidth for this QoS STA.
MISSING_ACKS	34	34	Disassociated because excessive number of frames need to be acknowledged, but are not acknowledged due to AP transmissions and/or poor channel conditions.
EXCEEDED_TXOP	35	35	Disassociated because STA is transmitting outside the limits of its TXOPs.
STA_LEAVING	36	36	Requesting STA is leaving the BSS (or resetting).
END_BA	37	37	Requesting STA is no longer using the stream or session.
UNKNOWN_BA	38	38	Requesting STA received frames using a mechanism for which a setup has not been completed.
TIMEOUT	39	39	Requested from peer STA due to timeout
Reserved	40 ~ 45	40 ~ 45	
PEER_INITIATED	46	46	In a Disassociation frame: Disassociated because authorized access limit reached.
AP_INITIATED	47	47	In a Disassociation frame: Disassociated due to external service requirements.
INVALID_FT_ACTION_FRAME_COUNT	48	48	Invalid FT Action frame count.

continues on next page

Table 30 – continued from previous page

Reason code	Value	Mapped To	Description
INVALID_PMKID	49	49	Invalid pairwise master key identifier (PMKID).
INVALID_MDE	50	50	Invalid MDE.
INVALID_FTE	51	51	Invalid FTE
TRANSMISSION_LINK_ESTABLISHMENT_FAILED	67	67	Transmission link establishment in alternative channel failed.
ALTERATIVE_CHANNEL_OCCUPIED	68	68	The alternative channel is occupied.
BEACON_TIMEOUT	200	reserved	Espressif-specific Wi-Fi reason code: when the station loses N beacons continuously, it will disrupt the connection and report this reason.
NO_AP_FOUND	201	reserved	Espressif-specific Wi-Fi reason code: when the station fails to scan the target AP, this reason code will be reported.
AUTH_FAIL	202	reserved	Espressif-specific Wi-Fi reason code: the authentication fails, but not because of a timeout.
ASSOC_FAIL	203	reserved	Espressif-specific Wi-Fi reason code: the association fails, but not because of ASSOC_EXPIRE or ASSOC_TOOMANY.
HANDSHAKE_TIMEOUT	204	reserved	Espressif-specific Wi-Fi reason code: the handshake fails for the same reason as that in WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT.
CONNECTION_FAIL	205	reserved	Espressif-specific Wi-Fi reason code: the connection to the AP has failed.

Wi-Fi Reason code related to wrong password

The table below shows the Wi-Fi reason-code may related to wrong password.

Reason code	Value	Description
4WAY_HANDSHAKE_TIMEOUT	15	Four-way handshake times out. Setting wrong password when STA connecting to an encrpyted AP.
NO_AP_FOUND	201	This may related to wrong password in the two scenarios: • Setting password when STA connecting to an unencrypted AP. • Doesn’t setting password when STA connecting to an encrypted AP.
HANDSHAKE_TIMEOUT	204	Four-way handshake fails.

Wi-Fi Reason code related to low RSSI

The table below shows the Wi-Fi reason-code may related to low RSSI.

Reason code	Value	Description
NO_AP_FOUND	201	The station fails to scan the target AP due to low RSSI
HANDSHAKE_TIMEOUT	204	Four-way handshake fails.

4.31.11 ESP32 Wi-Fi Station Connecting When Multiple APs Are Found

This scenario is similar as *ESP32 Wi-Fi Station Connecting Scenario*. The difference is that the station will not raise the event *WIFI_EVENT_STA_DISCONNECTED* unless it fails to connect all of the found APs.

4.31.12 Wi-Fi Reconnect

The station may disconnect due to many reasons, e.g., the connected AP is restarted. It is the application's responsibility to reconnect. The recommended reconnection strategy is to call `esp_wifi_connect()` on receiving event *WIFI_EVENT_STA_DISCONNECTED*.

Sometimes the application needs more complex reconnection strategy:

- If the disconnect event is raised because the `esp_wifi_disconnect()` is called, the application may not want to do the reconnection.
- If the `esp_wifi_scan_start()` may be called at anytime, a better reconnection strategy is necessary. Refer to *Scan When Wi-Fi Is Connecting*.

Another thing that need to be considered is that the reconnection may not connect the same AP if there are more than one APs with the same SSID. The reconnection always select current best APs to connect.

4.31.13 Wi-Fi Beacon Timeout

The beacon timeout mechanism is used by ESP32 station to detect whether the AP is alive or not. If the station does not receive the beacon of the connected AP within the inactive time, the beacon timeout happens. The application can set inactive time via API `esp_wifi_set_inactive_time()`.

After the beacon times out, the station sends 5 probe requests to the AP. If still no probe response or beacon is received from AP, the station disconnects from the AP and raises the event *WIFI_EVENT_STA_DISCONNECTED*.

It should be considered that the timer used for beacon timeout will be reset during the scanning process. It means that the scan process will affect the triggering of the event *WIFI_EVENT_STA_BEACON_TIMEOUT*.

4.31.14 ESP32 Wi-Fi Configuration

All configurations will be stored into flash when the Wi-Fi NVS is enabled; otherwise, refer to *Wi-Fi NVS Flash*.

Wi-Fi Mode

Call `esp_wifi_set_mode()` to set the Wi-Fi mode.

Mode	Description
WIFI_MODE_NULL	NULL mode: in this mode, the internal data struct is not allocated to the station and the AP, while both the station and AP interfaces are not initialized for RX/TX Wi-Fi data. Generally, this mode is used for Sniffer, or when you only want to stop both the station and the AP without calling `esp_wifi_deinit()` to unload the whole Wi-Fi driver.
WIFI_MODE_STA	Station mode: in this mode, `esp_wifi_start()` will init the internal station data, while the station's interface is ready for the RX and TX Wi-Fi data. After `esp_wifi_connect()`, the station will connect to the target AP.
WIFI_MODE_AP	AP mode: in this mode, `esp_wifi_start()` will init the internal AP data, while the AP's interface is ready for RX/TX Wi-Fi data. Then, the Wi-Fi driver starts broad-casting beacons, and the AP is ready to get connected to other stations.
WIFI_MODE_APSTA	Station/AP coexistence mode: in this mode, `esp_wifi_start()` will simultaneously init both the station and the AP. This is done in station mode and AP mode. Please note that the channel of the external AP, which the ESP station is connected to, has higher priority over the ESP AP channel.

Station Basic Configuration

API `esp_wifi_set_config()` can be used to configure the station. And the configuration will be stored in NVS. The table below describes the fields in detail.

Field	Description
ssid	This is the SSID of the target AP, to which the station wants to connect.
password	Password of the target AP.
scan_method	For WIFI_FAST_SCAN scan, the scan ends when the first matched AP is found. For WIFI_ALL_CHANNEL_SCAN, the scan finds all matched APs on all channels. The default scan is WIFI_FAST_SCAN.
bssid_set	If bssid_set is 0, the station connects to the AP whose SSID is the same as the field "ssid", while the field "bssid" is ignored. In all other cases, the station connects to the AP whose SSID is the same as the "ssid" field, while its BSSID is the same the "bssid" field .
bssid	This is valid only when bssid_set is 1; see field "bssid_set".
channel	If the channel is 0, the station scans the channel 1 ~ N to search for the target AP; otherwise, the station starts by scanning the channel whose value is the same as that of the "channel" field, and then scans the channel 1 ~ N but skip the specific channel to find the target AP. For example, if the channel is 3, the scan order will be 3, 1, 2, 4,⋯ , N. If you do not know which channel the target AP is running on, set it to 0.
sort_method	This field is only for WIFI_ALL_CHANNEL_SCAN. If the sort_method is WIFI_CONNECT_AP_BY_SIGNAL, all matched APs are sorted by signal, and the AP with the best signal will be connected firstly. For example, the station wants to connect an AP whose SSID is "apxx". If the scan finds two APs whose SSID equals to "apxx", and the first AP's signal is -90 dBm while the second AP's signal is -30 dBm, the station connects the second AP firstly, and it would not connect the first one unless it fails to connect the second one. If the sort_method is WIFI_CONNECT_AP_BY_SECURITY, all matched APs are sorted by security. For example, the station wants to connect an AP whose SSID is "apxx". If the scan finds two APs whose SSID is "apxx", and the security of the first found AP is open while the second one is WPA2, the station connects to the second AP firstly, and it would not connect the first one unless it fails to connect the second one.
threshold	The threshold is used to filter the found AP. If the RSSI or security mode is less than the configured threshold, the AP will be discarded. If the RSSI is set to 0, it means the default threshold and the default RSSI threshold are -127 dBm. If the authmode threshold is set to 0, it means the default threshold and the default authmode threshold are open.

Attention: WEP/WPA security modes are deprecated in IEEE 802.11-2016 specifications and are recommended not to be used. These modes can be rejected using authmode threshold by setting threshold as WPA2 by threshold.authmode as WIFI_AUTH_WPA2_PSK.

AP Basic Configuration

API `esp_wifi_set_config()` can be used to configure the AP. And the configuration will be stored in NVS. The table below describes the fields in detail.

Field	Description
ssid	SSID of AP; if the ssid[0] is 0xFF and ssid[1] is 0xFF, the AP defaults the SSID to ESP_aabbcc, where "aabbcc" is the last three bytes of the AP MAC.
password	Password of AP; if the auth mode is WIFI_AUTH_OPEN, this field will be ignored.
ssid_len	Length of SSID; if ssid_len is 0, check the SSID until there is a termination character. If ssid_len > 32, change it to 32; otherwise, set the SSID length according to ssid_len.
channel	Channel of AP; if the channel is out of range, the Wi-Fi driver defaults to channel 1. So, please make sure the channel is within the required range. For more details, refer to *Wi-Fi Country Code*.
authmode	Auth mode of ESP AP; currently, ESP AP does not support AUTH_WEP. If the authmode is an invalid value, AP defaults the value to WIFI_AUTH_OPEN.
ssid_hidden	If ssid_hidden is 1, AP does not broadcast the SSID; otherwise, it does broadcast the SSID.
max_connection	The max number of stations allowed to connect in, the default value is 10. ESP Wi-Fi supports up to 15 (ESP_WIFI_MAX_CONN_NUM) Wi-Fi connections. Please note that ESP AP and ESP-NOW share the same encryption hardware keys, so the max_connection parameter will be affected by the *CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM*. The total number of encryption hardware keys is 17, if *CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM* <= 2, the max_connection can be set up to 15, otherwise the max_connection can be set up to (17 - *CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM*).
beacon_interval	Beacon interval; the value is 100 ~ 60000 ms, with default value being 100 ms. If the value is out of range, AP defaults it to 100 ms.

Wi-Fi Protocol Mode

Currently, the ESP-IDF supports the following protocol modes:

Protocol Mode	Description
802.11b	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B) to set the station/AP to 802.11b-only mode.
802.11bg	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G) to set the station/AP to 802.11bg mode.
802.11g	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G) and esp_wifi_config_11b_rate(ifx, true) to set the station/AP to 802.11g mode.
802.11bgn	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11N) to set the station/ AP to BGN mode.
802.11gn	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11 and esp_wifi_config_11b_rate(ifx, true) to set the station/AP to 802.11gn mode.
802.11 BGNLR	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11N|WIFI_PROTOCOL_LR) to set the station/AP to BGN and the LR mode.
802.11 LR	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_LR) to set the station/AP only to the LR mode. **This mode is an Espressif-patented mode which can achieve a one-kilometer line of sight range. Please make sure both the station and the AP are connected to an ESP device.**

Long Range (LR)

Long Range (LR) mode is an Espressif-patented Wi-Fi mode which can achieve a one-kilometer line of sight range. Compared to the traditional 802.11b mode, it has better reception sensitivity, stronger anti-interference ability, and longer transmission distance.

LR Compatibility Since LR is Espressif-unique Wi-Fi mode, only ESP32 devices can transmit and receive the LR data. In other words, the ESP32 device should NOT transmit the data in LR data rate if the connected device does not support LR. The application can achieve this by configuring suitable Wi-Fi mode. If the negotiated mode supports LR, the ESP32 may transmit data in LR rate. Otherwise, ESP32 will transmit all data in traditional Wi-Fi data rate.

The following table depicts the Wi-Fi mode negotiation:

APSTA	BGN	BG	B	BGNLR	BGLR	BLR	LR
BGN	BGN	BG	B	BGN	BG	B	•
BG	BG	BG	B	BG	BG	B	•
B	B	B	B	B	B	B	•
BGNLR	•	•	•	BGNLR	BGLR	BLR	LR
BGLR	•	•	•	BGLR	BGLR	BLR	LR
BLR	•	•	•	BLR	BLR	BLR	LR
LR	•	•	•	LR	LR	LR	LR

In the above table, the row is the Wi-Fi mode of AP and the column is the Wi-Fi mode of station. The "-" indicates Wi-Fi mode of the AP and station are not compatible.

According to the table, the following conclusions can be drawn:

- For LR-enabled AP of ESP32, it is incompatible with traditional 802.11 mode, because the beacon is sent in LR mode.
- For LR-enabled station of ESP32 whose mode is NOT LR-only mode, it is compatible with traditional 802.11 mode.
- If both station and AP are ESP32 devices and both of them have enabled LR mode, the negotiated mode supports LR.

If the negotiated Wi-Fi mode supports both traditional 802.11 mode and LR mode, it is the Wi-Fi driver' s responsibility to automatically select the best data rate in different Wi-Fi modes and the application can ignore it.

LR Impacts to Traditional Wi-Fi Device The data transmission in LR rate has no impacts on the traditional Wi-Fi device because:

- The CCA and backoff process in LR mode are consistent with 802.11 specification.
- The traditional Wi-Fi device can detect the LR signal via CCA and do backoff.

In other words, the transmission impact in LR mode is similar to that in 802.11b mode.

LR Transmission Distance The reception sensitivity gain of LR is about 4 dB larger than that of the traditional 802.11b mode. Theoretically, the transmission distance is about 2 to 2.5 times the distance of 11B.

LR Throughput The LR rate has very limited throughput, because the raw PHY data rate LR is 1/2 Mbps and 1/4 Mbps.

When to Use LR The general conditions for using LR are:

- Both the AP and station are Espressif devices.
- Long distance Wi-Fi connection and data transmission is required.
- Data throughput requirements are very small, such as remote device control.

Wi-Fi Country Code

Call `esp_wifi_set_country()` to set the country info. The table below describes the fields in detail. Please consult local 2.4 GHz RF operating regulations before configuring these fields.

Field	Description
cc[3]	Country code string. This attribute identifies the country or noncountry entity in which the station/AP is operating. If it is a country, the first two octets of this string is the two-character country info as described in the document ISO/IEC3166-1. The third octect is one of the following: • an ASCII space character, which means the regulations under which the station/AP is operating encompass all environments for the current frequency band in the country. • an ASCII ‘O’ character, which means the regulations under which the station/AP is operating are for an outdoor environment only. • an ASCII ‘I’ character, which means the regulations under which the station/AP is operating are for an indoor environment only. • an ASCII ‘X’ character, which means the station/AP is operating under a non-country entity. The first two octets of the noncountry entity is two ASCII ‘XX’ characters. • the binary representation of the Operating Class table number currently in use. Refer to Annex E of IEEE Std 802.11-2020.
schan	Start channel. It is the minimum channel number of the regulations under which the station/AP can operate.
nchan	Total number of channels as per the regulations. For example, if the schan=1, nchan=13, then the station/AP can send data from channel 1 to 13.
policy	Country policy. This field controls which country info will be used if the configured country info is in conflict with the connected AP’s. For more details on related policies, see the following section.

The default country info is:

```
wifi_country_t config = {
    .cc = "01",
    .schan = 1,
    .nchan = 11,
    .policy = WIFI_COUNTRY_POLICY_AUTO,
};
```

If the Wi-Fi Mode is station/AP coexist mode, they share the same configured country info. Sometimes, the country info of AP, to which the station is connected, is different from the country info of configured. For example, the configured station has country info:

```
wifi_country_t config = {
    .cc = "JP",
    .schan = 1,
    .nchan = 14,
    .policy = WIFI_COUNTRY_POLICY_AUTO,
};
```

but the connected AP has country info:

```
wifi_country_t config = {
    .cc = "CN",
    .schan = 1,
    .nchan = 13,
};
```

then country info of connected AP’s is used.

The following table depicts which country info is used in different Wi-Fi modes and different country policies, and it also describes the impact on active scan.

Wi-Fi Mode	Policy	Description
Station	WIFI_COUNTRY_POLICY_AUTO	If the connected AP has country IE in its beacon, the country info equals to the country info in beacon. Otherwise, use the default country info. For scan: Use active scan from 1 to 11 and use passive scan from 12 to 14. Always keep in mind that if an AP with hidden SSID and station is set to a passive scan channel, the passive scan will not find it. In other words, if the application hopes to find the AP with hidden SSID in every channel, the policy of country info should be configured to WIFI_COUNTRY_POLICY_MANUAL.
Station	WIFI_COUNTRY_POLICY_MANUAL	Always use the configured country info. For scan: Use active scan from schan to schan+nchan-1.
AP	WIFI_COUNTRY_POLICY_AUTO	Always use the configured country info.
AP	WIFI_COUNTRY_POLICY_MANUAL	Always use the configured country info.
Station/AP-coexistence	WIFI_COUNTRY_POLICY_AUTO	Station: Same as station mode with policy WIFI_COUNTRY_POLICY_AUTO. AP: If the station does not connect to any external AP, the AP uses the configured country info. If the station connects to an external AP, the AP has the same country info as the station.
Station/AP-coexistence	WIFI_COUNTRY_POLICY_MANUAL	Station: Same as station mode with policy WIFI_COUNTRY_POLICY_MANUAL. AP: Same as AP mode with policy WIFI_COUNTRY_POLICY_MANUAL.

Home Channel In AP mode, the home channel is defined as the AP channel. In station mode, home channel is defined as the channel of AP which the station is connected to. In station/AP-coexistence mode, the home channel of AP and station must be the same, and if they are different, the station's home channel is always in priority. For example, assume that the AP is on channel 6, and the station connects to an AP whose channel is 9. Since the station's home channel has higher priority, the AP needs to switch its channel from 6 to 9 to make sure that it has the same home channel as the station. While switching channel, the ESP32 in AP mode will notify the connected stations about the channel migration using a Channel Switch Announcement (CSA). Station that supports channel switching will transit without disconnecting and reconnecting to the AP.

Wi-Fi Vendor IE Configuration

By default, all Wi-Fi management frames are processed by the Wi-Fi driver, and the application can ignore them. However, some applications may have to handle the beacon, probe request, probe response, and other management frames. For example, if you insert some vendor-specific IE into the management frames, it is only the management frames which contain this vendor-specific IE that will be processed. In ESP32, `esp_wifi_set_vendor_ie()` and `esp_wifi_set_vendor_ie_cb()` are responsible for this kind of tasks.

4.31.15 Wi-Fi Easy Connect™ (DPP)

Wi-Fi Easy Connect™ (or Device Provisioning Protocol) is a secure and standardized provisioning protocol for configuring Wi-Fi devices. More information can be found in *esp_dpp*.

WPA2-Enterprise

WPA2-Enterprise is the secure authentication mechanism for enterprise wireless networks. It uses RADIUS server for authentication of network users before connecting to the Access Point. The authentication process is based on 802.1X policy and comes with different Extended Authentication Protocol (EAP) methods such as TLS, TTLS, and PEAP. RADIUS server authenticates the users based on their credentials (username and password), digital certificates, or both. When ESP32 in station mode tries to connect an AP in enterprise mode, it sends authentication request to AP which is sent to RADIUS server by AP for authenticating the station. Based on different EAP methods, the parameters can be set in configuration which can be opened using `idf.py menuconfig`. WPA2_Enterprise is supported by ESP32 only in station mode.

For establishing a secure connection, AP and station negotiate and agree on the best possible cipher suite to be used. ESP32 supports 802.1X/EAP (WPA) method of AKM and Advanced encryption standard with Counter Mode Cipher Block Chaining Message Authentication protocol (AES-CCM) cipher suite. It also supports the cipher suites supported by mbedtls if *USE_MBEDTLS_CRYPTO* flag is set.

ESP32 currently supports the following EAP methods:

- EAP-TLS: This is a certificate-based method and only requires SSID and EAP-IDF.
- PEAP: This is a Protected EAP method. Username and Password are mandatory.
- **EAP-TTLS: This is a credential-based method. Only server authentication is mandatory while user authenticatio**
  - PAP: Password Authentication Protocol.
  - CHAP: Challenge Handshake Authentication Protocol.
  - MSCHAP and MSCHAP-V2.
- EAP-FAST: This is an authentication method based on Protected Access Credentials (PAC) which also uses identity and password. Currently, USE_MBEDTLS_CRYPTO flag should be disabled to use this feature.

Detailed information on creating certificates and how to run wpa2_enterprise example on ESP32 can be found in wifi/wifi_enterprise.

4.31.16 Wireless Network Management

Wireless Network Management allows client devices to exchange information about the network topology, including information related to RF environment. This makes each client network-aware, facilitating overall improvement in the performace of the wireless network. It is part of 802.11v specification. It also enables the client to support Network assisted Roaming. - Network assisted Roaming: Enables WLAN to send messages to associated clients, resulting clients to associate with APs with better link metrics. This is useful for both load balancing and in directing poorly connected clients.

Current implementation of 802.11v includes support for BSS transition management frames.

4.31.17 Radio Resource Measurement

Radio Resource Measurement (802.11k) is intended to improve the way traffic is distributed within a network. In a WLAN, each device normally connects to the access point (AP) that provides the strongest signal. Depending on the number and geographic locations of the subscribers, this arrangement can sometimes lead to excessive demand on one AP and underutilization of others, resulting in degradation of overall network performance. In a network conforming to 802.11k, if the AP having the strongest signal is loaded to its full capacity, a wireless device can be moved to one of the underutilized APs. Even though the signal may be weaker, the overall throughput is greater because more efficient use is made of the network resources.

Current implementation of 802.11k includes support for beacon measurement report, link measurement report, and neighbor request.

Refer ESP-IDF example examples/wifi/roaming/README.md to set up and use these APIs. Example code only demonstrates how these APIs can be used, and the application should define its own algorithm and cases as required.

4.31.18 Fast BSS Transition

Fast BSS transition (802.11R FT), is a standard to permit continuous connectivity aboard wireless devices in motion, with fast and secure client transitions from one Basic Service Set (abbreviated BSS, and also known as a base station or more colloquially, an access point) to another performed in a nearly seamless manner **avoiding 802.1i 4 way handshake** . 802.11R specifies transitions between access points by redefining the security key negotiation protocol, allowing both the negotiation and requests for wireless resources to occur in parallel. The key derived from the server to be cached in the wireless network, so that a reasonable number of future connections can be based on the cached key, avoiding the 802.1X process

ESP32 station supports FT for WPA2-PSK networks. Do note that ESP32 station only support FT over the air protocol only.

A config option *CONFIG_WPA_11R_SUPPORT* and configuration parameter `ft_enabled` in `wifi_sta_config_t` is provided to enable 802.11R support for station. Refer ESP-IDF example examples/wifi/roaming/README.md for further details.

4.31.19 ESP32 Wi-Fi Power-saving Mode

This subsection will briefly introduce the concepts and usage related to Wi-Fi Power Saving Mode, for a more detailed introduction please refer to the *Low Power Mode User Guide*

Station Sleep

Currently, ESP32 Wi-Fi supports the Modem-sleep mode which refers to the legacy power-saving mode in the IEEE 802.11 protocol. Modem-sleep mode works in station-only mode and the station must connect to the AP first. If the Modem-sleep mode is enabled, station will switch between active and sleep state periodically. In sleep state, RF, PHY and BB are turned off in order to reduce power consumption. Station can keep connection with AP in modem-sleep mode.

Modem-sleep mode includes minimum and maximum power-saving modes. In minimum power-saving mode, station wakes up every DTIM to receive beacon. Broadcast data will not be lost because it is transmitted after DTIM. However, it cannot save much more power if DTIM is short for DTIM is determined by AP.

In maximum power-saving mode, station wakes up in every listen interval to receive beacon. This listen interval can be set to be longer than the AP DTIM period. Broadcast data may be lost because station may be in sleep state at DTIM time. If listen interval is longer, more power is saved, but broadcast data is more easy to lose. Listen interval can be configured by calling API `esp_wifi_set_config()` before connecting to AP.

Call `esp_wifi_set_ps(WIFI_PS_MIN_MODEM)` to enable Modem-sleep minimum power-saving mode or `esp_wifi_set_ps(WIFI_PS_MAX_MODEM)` to enable Modem-sleep maximum power-saving mode after calling `esp_wifi_init()`. When station connects to AP, Modem-sleep will start. When station disconnects from AP, Modem-sleep will stop.

Call `esp_wifi_set_ps(WIFI_PS_NONE)` to disable Modem-sleep entirely. This has much higher power consumption, but provides minimum latency for receiving Wi-Fi data in real time. When Modem-sleep is enabled, received Wi-Fi data can be delayed for as long as the DTIM period (minimum power-saving mode) or the listen interval (maximum power-saving mode).

Notice that at coexist mode, Wi-Fi would still keep active state in Wi-Fi time slice and only keep sleep state in non Wi-Fi time slice even when calling “esp_wifi_set_ps(WIFI_PS_NONE)” . Please refer to *coexist policy*.

The default Modem-sleep mode is WIFI_PS_MIN_MODEM.

AP Sleep

Currently, ESP32 AP does not support all of the power-saving feature defined in Wi-Fi specification. To be specific, the AP only caches unicast data for the stations connect to this AP, but does not cache the multicast data for the stations. If stations connected to the ESP32 AP are power-saving enabled, they may experience multicast packet loss.

In the future, all power-saving features will be supported on ESP32 AP.

Disconnected State Sleep

Disconnected state is the duration without Wi-Fi connection between `esp_wifi_start()` to `esp_wifi_stop()`.

Currently, ESP32 Wi-Fi supports sleep mode in disconnected state if running at station mode. This feature could be configured by Menuconfig choice *CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE*.

If *CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE* is enabled, RF, PHY and BB would be turned off in disconnected state when IDLE. The current would be same with current at modem-sleep.

The choice *CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE* would be selected by default, while it would be selected forcefully in Menuconfig at coexistence mode.

Connectionless Modules Power-saving

Connectionless modules are those Wi-Fi modules not relying on Wi-Fi connection, e.g ESP-NOW, DPP, FTM. These modules start from `esp_wifi_start()`, working until `esp_wifi_stop()`.

Currently, if ESP-NOW works at station mode, its supported to sleep at both connected state and disconnected state.

Connectionless Modules TX For each connectionless module, its supported to TX at any sleeping time without any extra configuration.

Meanwhile, `esp_wifi_80211_tx()` is supported at sleep as well.

Connectionless Modules RX For each connectionless module, two parameters shall be configured to RX at sleep, which are *Window* and *Interval*.

At the start of *Interval* time, RF, PHY, BB would be turned on and kept for *Window* time. Connectionless Module could RX in the duration.

Interval

- There is only one *Interval*. Its configured by `esp_wifi_set_connectionless_interval()`. The unit is milliseconds.
- The default value of *Interval* is *ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE*.
- Event *WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START* would be posted at the start of *Interval*. Since *Window* also starts at that momemt, its recommended to TX in that event.
- At connected state, the start of *Interval* would be aliged with TBTT.

Window

- Each connectionless module has its own *Window* after start. Connectionless Modules Power-saving would work with the max one among them.
- *Window* is configured by `module_name_set_wake_window()`. The unit is milliseconds.
- The default value of *Window* is the maximum.

Table 31: RF, PHY and BB usage under different circumstances

		Interval	
		ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE	
Window	0	not used	
	1 - maximum	default mode	used periodically (Window < Interval) / used all time (Window ≥ Interval)

Default mode If *Interval* is *ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE* with non-zero *Window*, Connectionless Modules Power-saving would work in default mode.

In default mode, RF, PHY, BB would be kept on if no coexistence with non-Wi-Fi protocol.

With coexistence, RF, PHY, BB resources are allocated by coexistence module to Wi-Fi connectionless module and non-Wi-Fi module, using time-division method. In default mode, Wi-Fi connectionless module is allowed to use RF, BB, PHY periodically under a stable performance.

Its recommended to configure Connectionless Modules Power-saving to default mode if there is Wi-Fi connectionless module coexists with non-Wi-Fi module.

4.31.20 ESP32 Wi-Fi Throughput

The table below shows the best throughput results gained in Espressif's lab and in a shielded box.

Type/Throughput	Air In Lab	Shield-box	Test Tool	IDF Version (commit ID)
Raw 802.11 Packet RX	N/A	**130 MBit/s**	Internal tool	NA
Raw 802.11 Packet TX	N/A	**130 MBit/s**	Internal tool	NA
UDP RX	30 MBit/s	85 MBit/s	iperf example	15575346
UDP TX	30 MBit/s	75 MBit/s	iperf example	15575346
TCP RX	20 MBit/s	65 MBit/s	iperf example	15575346
TCP TX	20 MBit/s	75 MBit/s	iperf example	15575346

When the throughput is tested by iperf example, the sdkconfig is examples/wifi/iperf/sdkconfig.defaults.esp32.

4.31.21 Wi-Fi 80211 Packet Send

The *esp_wifi_80211_tx()* API can be used to:

- Send the beacon, probe request, probe response, and action frame.
- Send the non-QoS data frame.

It cannot be used for sending encrypted or QoS frames.

Preconditions of Using `esp_wifi_80211_tx()`

- The Wi-Fi mode is station, or AP, or station/AP.
- Either esp_wifi_set_promiscuous(true), or *esp_wifi_start()*, or both of these APIs return ESP_OK. This is because Wi-Fi hardware must be initialized before *esp_wifi_80211_tx()* is called. In ESP32, both esp_wifi_set_promiscuous(true) and *esp_wifi_start()* can trigger the initialization of Wi-Fi hardware.
- The parameters of *esp_wifi_80211_tx()* are hereby correctly provided.

Data Rate

- The default data rate is 1 Mbps.
- Can set any rate through *esp_wifi_config_80211_tx_rate()* API.
- Can set any bandwidth through *esp_wifi_set_bandwidth()* API.

Side-Effects to Avoid in Different Scenarios

Theoretically, if the side-effects the API imposes on the Wi-Fi driver or other stations/APs are not considered, a raw 802.11 packet can be sent over the air with any destination MAC, any source MAC, any BSSID, or any other types of packet. However, robust or useful applications should avoid such side-effects. The table below provides some tips and recommendations on how to avoid the side-effects of *esp_wifi_80211_tx()* in different scenarios.

Scenario	Description
No Wi-Fi connection	In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If en_sys_seq==true, the Wi-Fi driver is responsible for the sequence control. If en_sys_seq==false, the application needs to ensure that the buffer has the correct sequence. Theoretically, the MAC address can be any address. However, this may impact other stations/APs with the same MAC/BSSID. Side-effect example#1 The application calls *esp_wifi_80211_tx()* to send a beacon with BSSID == mac_x in AP mode, but the mac_x is not the MAC of the AP interface. Moreover, there is another AP, e.g., "other-AP" , whose BSSID is mac_x. If this happens, an "unexpected behavior"may occur, because the stations which connect to the "other-AP"cannot figure out whether the beacon is from the "other-AP" or the *esp_wifi_80211_tx()*. To avoid the above-mentioned side-effects, it is recommended that: • If esp_wifi_80211_tx is called in station mode, the first MAC should be a multicast MAC or the exact target-device's MAC, while the second MAC should be that of the station interface. • If esp_wifi_80211_tx is called in AP mode, the first MAC should be a multicast MAC or the exact target-device's MAC, while the second MAC should be that of the AP interface. The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons.
Have Wi-Fi connection	When the Wi-Fi connection is already set up, and the sequence is controlled by the application, the latter may impact the sequence control of the Wi-Fi connection as a whole. So, the `en_sys_seq` need to be true, otherwise `ESP_ERR_INVALID_ARG` is returned. The MAC-address recommendations in the "No Wi-Fi connection" scenario also apply to this scenario. If the Wi-Fi mode is station mode, the MAC address1 is the MAC of AP to which the station is connected, and the MAC address2 is the MAC of station interface, it is said that the packet is sent from the station to AP. Otherwise, if the Wi-Fi is in AP mode, the MAC address1 is the MAC of the station that connects to this AP, and the MAC address2 is the MAC of AP interface, it is said that the packet is sent from the AP to station. To avoid conflicting with Wi-Fi connections, the following checks are applied: • If the packet type is data and is sent from the station to AP, the ToDS bit in IEEE 80211 frame control should be 1 and the FromDS bit should be 0. Otherwise, the packet will be discarded by Wi-Fi driver. • If the packet type is data and is sent from the AP to station, the ToDS bit in IEEE 80211 frame control should be 0 and the FromDS bit should be 1. Otherwise, the packet will be discarded by Wi-Fi driver. • If the packet is sent from station to AP or from AP to station, the Power Management, More Data, and Re-Transmission bits should be 0. Otherwise, the packet will be discarded by Wi-Fi driver. `ESP_ERR_INVALID_ARG` is returned if any check fails.

4.31.22 Wi-Fi Sniffer Mode

The Wi-Fi sniffer mode can be enabled by esp_wifi_set_promiscuous(). If the sniffer mode is enabled, the following packets **can** be dumped to the application:

- 802.11 Management frame.
- 802.11 Data frame, including MPDU, AMPDU, and AMSDU.
- 802.11 MIMO frame, for MIMO frame, the sniffer only dumps the length of the frame.

- 802.11 Control frame.
- 802.11 CRC error frame.

The following packets will **NOT** be dumped to the application:

- Other 802.11 error frames.

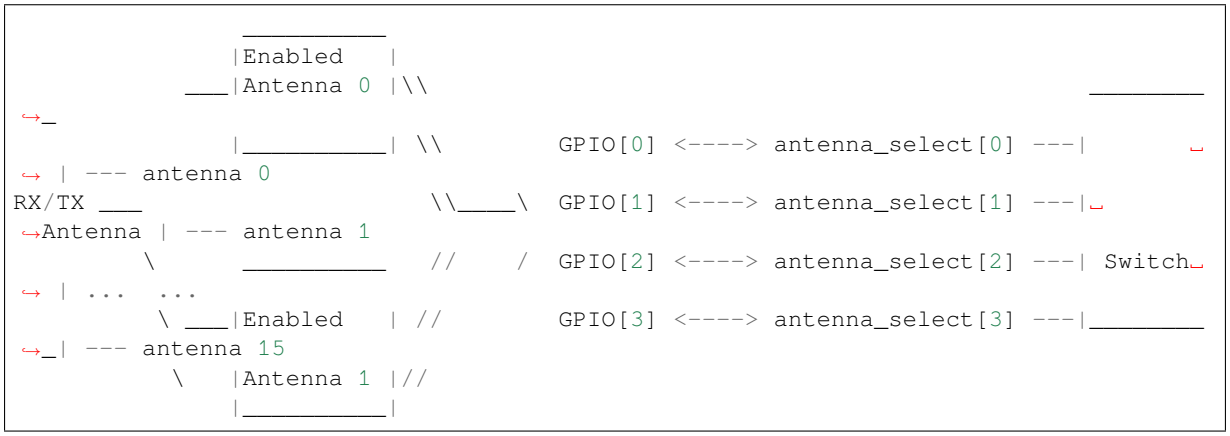
For frames that the sniffer **can** dump, the application can additionally decide which specific type of packets can be filtered to the application by using *esp_wifi_set_promiscuous_filter()* and *esp_wifi_set_promiscuous_ctrl_filter()*. By default, it will filter all 802.11 data and management frames to the application. If you want to filter the 802.11 control frames, the filter parameter in *esp_wifi_set_promiscuous_filter()* should include *WIFI_PROMIS_FILTER_MASK_CTRL* type, and if you want to differentiate control frames further, then call *esp_wifi_set_promiscuous_ctrl_filter()*.

The Wi-Fi sniffer mode can be enabled in the Wi-Fi mode of *WIFI_MODE_NULL*, *WIFI_MODE_STA*, *WIFI_MODE_AP*, or *WIFI_MODE_APSTA*. In other words, the sniffer mode is active when the station is connected to the AP, or when the AP has a Wi-Fi connection. Please note that the sniffer has a **great impact** on the throughput of the station or AP Wi-Fi connection. Generally, the sniffer should be enabled **only if** the station/AP Wi-Fi connection does not experience heavy traffic.

Another noteworthy issue about the sniffer is the callback wifi_promiscuous_cb_t. The callback will be called directly in the Wi-Fi driver task, so if the application has a lot of work to do for each filtered packet, the recommendation is to post an event to the application task in the callback and defer the real work to the application task.

4.31.23 Wi-Fi Multiple Antennas

The Wi-Fi multiple antennas selecting can be depicted as following picture:



ESP32 supports up to sixteen antennas through external antenna switch. The antenna switch can be controlled by up to four address pins - antenna_select[0:3]. Different input value of antenna_select[0:3] means selecting different antenna. For example, the value ‘0b1011’ means the antenna 11 is selected. The default value of antenna_select[3:0] is ‘0b0000’, which means the antenna 0 is selected by default.

Up to four GPIOs are connected to the four active high antenna_select pins. ESP32 can select the antenna by control the GPIO[0:3]. The API *esp_wifi_set_ant_gpio()* is used to configure which GPIOs are connected to antenna_selects. If GPIO[x] is connected to antenna_select[x], then gpio_config->gpio_cfg[x].gpio_select should be set to 1 and gpio_config->gpio_cfg[x].gpio_num should be provided.

For the specific implementation of the antenna switch, there may be illegal values in *antenna_select[0:3]*. It means that ESP32 may support less than sixteen antennas through the switch. For example, ESP32-WROOM-DA which uses RTC6603SP as the antenna switch, supports two antennas. Two GPIOs are connected to two active high antenna selection inputs. The value ‘0b01’ means the antenna 0 is selected, the value ‘0b10’ means the antenna 1 is selected. Values ‘0b00’ and ‘0b11’ are illegal.

Although up to sixteen antennas are supported, only one or two antennas can be simultaneously enabled for RX/TX. The API *esp_wifi_set_ant()* is used to configure which antennas are enabled.

The enabled antennas selecting algorithm is also configured by *esp_wifi_set_ant()*. The RX/TX antenna mode can be *WIFI_ANT_MODE_ANT0*, *WIFI_ANT_MODE_ANT1*, or *WIFI_ANT_MODE_AUTO*. If the antenna mode is *WIFI_ANT_MODE_ANT0*, the enabled antenna 0 is selected for RX/TX data. If the antenna mode is *WIFI_ANT_MODE_ANT1*, the enabled antenna 1 is selected for RX/TX data. Otherwise, Wi-Fi automatically selects the enabled antenna that has better signal.

If the RX antenna mode is *WIFI_ANT_MODE_AUTO*, the default antenna mode also needs to be set, because the RX antenna switching only happens when some conditions are met. For example, the RX antenna starts to switch if the RSSI is lower than -65 dBm or another antenna has better signal. RX uses the default antenna if the conditions are not met. If the default antenna mode is *WIFI_ANT_MODE_ANT1*, the enabled antenna 1 is used as the default RX antenna, otherwise the enabled antenna 0 is used.

Some limitations need to be considered:

- The TX antenna can be set to *WIFI_ANT_MODE_AUTO* only if the RX antenna mode is *WIFI_ANT_MODE_AUTO*, because TX antenna selecting algorithm is based on RX antenna in *WIFI_ANT_MODE_AUTO* type.
- When the TX antenna mode or RX antenna mode is configured to *WIFI_ANT_MODE_AUTO* the switching mode will easily trigger the switching phase, as long as there is deterioration of the RF signal. So in situations where the RF signal is not stable, the antenna switching will occur frequently, resulting in an RF performance that may not meet expectations.
- Currently, Bluetooth® does not support the multiple antennas feature, so please do not use multiple antennas related APIs.

Following is the recommended scenarios to use the multiple antennas:

- The applications can always choose to select a specified antenna or implement their own antenna selecting algorithm, e.g., selecting the antenna mode based on the information collected by the application. Refer to ESP-IDF example examples/wifi/antenna/README.md for the antenna selecting algorithm design.
- Both RX/TX antenna modes are configured to WIFI_ANT_MODE_ANT0 or WIFI_ANT_MODE_ANT1.

Wi-Fi Multiple Antennas Configuration

Generally, following steps can be taken to configure the multiple antennas:

- Configure which GPIOs are connected to the antenna_selects. For example, if four antennas are supported and GPIO20/GPIO21 are connected to antenna_select[0]/antenna_select[1], the configurations look like:

```
wifi_ant_gpio_config_t ant_gpio_config = {
    .gpio_cfg[0] = { .gpio_select = 1, .gpio_num = 20 },
    .gpio_cfg[1] = { .gpio_select = 1, .gpio_num = 21 }
};
```

- Configure which antennas are enabled and how RX/TX use the enabled antennas. For example, if antenna1 and antenna3 are enabled, the RX needs to select the better antenna automatically and uses antenna1 as its default antenna, the TX always selects the antenna3. The configuration looks like:

```
wifi_ant_config_t config = {
    .rx_ant_mode = WIFI_ANT_MODE_AUTO,
    .rx_ant_default = WIFI_ANT_ANT0,
    .tx_ant_mode = WIFI_ANT_MODE_ANT1,
    .enabled_ant0 = 1,
    .enabled_ant1 = 3
};
```

4.31.24 Wi-Fi Channel State Information

Channel state information (CSI) refers to the channel information of a Wi-Fi connection. In ESP32, this information consists of channel frequency responses of sub-carriers and is estimated when packets are received from the transmitter. Each channel frequency response of sub-carrier is recorded by two bytes of signed characters. The first one

is imaginary part and the second one is real part. There are up to three fields of channel frequency responses according to the type of received packet. They are legacy long training field (LLTF), high throughput LTF (HT-LTF), and space time block code HT-LTF (STBC-HT-LTF). For different types of packets which are received on channels with different state, the sub-carrier index and total bytes of signed characters of CSI are shown in the following table.

channel	secondary channel	none			below					above				
packet information	signal mode	non HT	HT		non HT	HT				non HT	HT			
	channel bandwidth	20 MHz	20 MHz		20 MHz	20 MHz		40 MHz		20 MHz	20 MHz		40 MHz	
	STBC	non STBC	non STBC	STBC	non STBC	non STBC	STBC	non STBC	STBC	non STBC	non STBC	STBC	non STBC	STBC
sub-carrier index	LLTF	0~31, -32~-1	0~31, -32~-1	0~31, -32~-1	0~63	0~63	0~63	0~63	0~63	-64~-1	-64~-1	-64~-1	-64~-1	-64~-1
	HT-LTF	•	0~31, -32~-1	0~31, -32~-1	•	0~63	0~62	0~63, -64~-1	0~60, -60~-1	•	-64~-1	-62~-1	0~63, -64~-1	0~60, -60~-1
	STBC-HT-LTF	•	•	0~31, -32~-1	•	•	0~62	•	0~60, -60~-1	•	•	-62~-1	•	0~60, -60~-1
total bytes		128	256	384	128	256	380	384	612	128	256	376	384	612

All of the information in the table can be found in the structure wifi_csi_info_t.

- Secondary channel refers to secondary_channel field of rx_ctrl field.
- Signal mode of packet refers to sig_mode field of rx_ctrl field.
- Channel bandwidth refers to cwb field of rx_ctrl field.
- STBC refers to stbc field of rx_ctrl field.
- Total bytes refers to len field.
- The CSI data corresponding to each Long Training Field (LTF) type is stored in a buffer starting from the buf field. Each item is stored as two bytes: imaginary part followed by real part. The order of each item is the same as the sub-carrier in the table. The order of LTF is: LLTF, HT-LTF, STBC-HT-LTF. However, all 3 LTFs may not be present, depending on the channel and packet information (see above).
- If first_word_invalid field of wifi_csi_info_t is true, it means that the first four bytes of CSI data is invalid due to a hardware limitation in ESP32.
- More information like RSSI, noise floor of RF, receiving time and antenna is in the rx_ctrl field.

When imaginary part and real part data of sub-carrier are used, please refer to the table below.

PHY standard	Sub-carrier range	Pilot sub-carrier	Sub-carrier(total/data)
802.11a/g	-26 to +26	-21, -7, +7, +21	52 total, 48 usable
802.11n, 20MHz	-28 to +28	-21, -7, +7, +21	56 total, 52 usable
802.11n, 40MHz	-57 to +57	-53, -25, -11, +11, +25, +53	114 total, 108 usable

Note:

- For STBC packet, CSI is provided for every space-time stream without CSD (cyclic shift delay). As each cyclic shift on the additional chains shall be -200 ns, only the CSD angle of first space-time stream is recorded

in sub-carrier 0 of HT-LTF and STBC-HT-LTF for there is no channel frequency response in sub-carrier 0. CSD[10:0] is 11 bits, ranging from -pi to pi.
- If LLTF, HT-LTF, or STBC-HT-LTF is not enabled by calling API *esp_wifi_set_csi_config()*, the total bytes of CSI data will be fewer than that in the table. For example, if LLTF and HT-LTF is not enabled and STBC-HT-LTF is enabled, when a packet is received with the condition above/HT/40MHz/STBC, the total bytes of CSI data is 244 ((61 + 60) * 2 + 2 = 244. The result is aligned to four bytes, and the last two bytes are invalid).

4.31.25 Wi-Fi Channel State Information Configure

To use Wi-Fi CSI, the following steps need to be done.

- Select Wi-Fi CSI in menuconfig. Go to `Menuconfig > Components config > Wi-Fi > Wi-Fi CSI (Channel State Information)`.
- Set CSI receiving callback function by calling API *esp_wifi_set_csi_rx_cb()*.
- Configure CSI by calling API *esp_wifi_set_csi_config()*.
- Enable CSI by calling API *esp_wifi_set_csi()*.

The CSI receiving callback function runs from Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post necessary data to a queue and handle it from a lower priority task. Because station does not receive any packet when it is disconnected and only receives packets from AP when it is connected, it is suggested to enable sniffer mode to receive more CSI data by calling *esp_wifi_set_promiscuous()*.

4.31.26 Wi-Fi HT20/40

ESP32 supports Wi-Fi bandwidth HT20 or HT40 and does not support HT20/40 coexist. *esp_wifi_set_bandwidth()* can be used to change the default bandwidth of station or AP. The default bandwidth for ESP32 station and AP is HT40.

In station mode, the actual bandwidth is firstly negotiated during the Wi-Fi connection. It is HT40 only if both the station and the connected AP support HT40, otherwise it is HT20. If the bandwidth of connected AP is changes, the actual bandwidth is negotiated again without Wi-Fi disconnecting.

Similarly, in AP mode, the actual bandwidth is negotiated between AP and the stations that connect to the AP. It is HT40 if the AP and one of the stations support HT40, otherwise it is HT20.

In station/AP coexist mode, the station/AP can configure HT20/40 seperately. If both station and AP are negotiated to HT40, the HT40 channel should be the channel of station because the station always has higher priority than AP in ESP32. For example, the configured bandwidth of AP is HT40, the configured primary channel is 6, and the configured secondary channel is 10. The station is connected to an router whose primary channel is 6 and secondary channel is 2, then the actual channel of AP is changed to primary 6 and secondary 2 automatically.

Theoretically, the HT40 can gain better throughput because the maximum raw physicial (PHY) data rate for HT40 is 150 Mbps while it is 72 Mbps for HT20. However, if the device is used in some special environment, e.g., there are too many other Wi-Fi devices around the ESP32 device, the performance of HT40 may be degraded. So if the applications need to support same or similar scenarios, it is recommended that the bandwidth is always configured to HT20.

4.31.27 Wi-Fi QoS

ESP32 supports all the mandatory features required in WFA Wi-Fi QoS Certification.

Four ACs (Access Category) are defined in Wi-Fi specification, and each AC has its own priority to access the Wi-Fi channel. Moreover, a map rule is defined to map the QoS priority of other protocol, e.g., 802.11D or TCP/IP precedence is mapped to Wi-Fi AC.

The table below describes how the IP Precedences are mapped to Wi-Fi ACs in ESP32. It also indicates whether the AMPDU is supported for this AC. The table is sorted from high to low priority. That is to say, the AC_VO has the highest priority.

IP Precedence	Wi-Fi AC	Support AMPDU?
6, 7	AC_VO (Voice)	No
4, 5	AC_VI (Video)	Yes
3, 0	AC_BE (Best Effort)	Yes
1, 2	AC_BK (Background)	Yes

The application can make use of the QoS feature by configuring the IP precedence via socket option IP_TOS. Here is an example to make the socket to use VI queue:

```
const int ip_precedence_vi = 4;
const int ip_precedence_offset = 5;
int priority = (ip_precedence_vi << ip_precedence_offset);
setsockopt(socket_id, IPPROTO_IP, IP_TOS, &priority, sizeof(priority));
```

Theoretically, the higher priority AC has better performance than the lower priority AC. However, it is not always true. Here are some suggestions about how to use the Wi-Fi QoS:

- Some really important application traffic can be put into the AC_VO queue. But avoid using the AC_VO queue for heavy traffic, as it may impact the management frames which also use this queue. Eventually, it is worth noting that the AC_VO queue does not support AMPDU, and its performance with heavy traffic is no better than other queues.
- Avoid using more than two precedences supported by different AMPDUs, e.g., when socket A uses precedence 0, socket B uses precedence 1, and socket C uses precedence 2. This can be a bad design because it may need much more memory. To be specific, the Wi-Fi driver may generate a Block Ack session for each precedence and it needs more memory if the Block Ack session is set up.

4.31.28 Wi-Fi AMSDU

ESP32 supports receiving and transmitting AMSDU. AMSDU TX is disabled by default, since enable AMSDU TX need more memory. Select *CONFIG_ESP32_WIFI_AMSDU_TX_ENABLED* to enable AMSDU Tx feature, it depends on *CONFIG_SPIRAM*.

4.31.29 Wi-Fi Fragment

supports Wi-Fi receiving fragment, but does not support Wi-Fi transmitting fragment.

4.31.30 WPS Enrollee

ESP32 supports WPS enrollee feature in Wi-Fi mode `WIFI_MODE_STA` or `WIFI_MODE_APSTA`. Currently, ESP32 supports WPS enrollee type PBC and PIN.

4.31.31 Wi-Fi Buffer Usage

This section is only about the dynamic buffer configuration.

Why Buffer Configuration Is Important

In order to get a high-performance system, consider the memory usage/configuration carefully for the following reasons:

- the available memory in ESP32 is limited.

- currently, the default type of buffer in LwIP and Wi-Fi drivers is "dynamic", **which means that both the LwIP and Wi-Fi share memory with the application**. Programmers should always keep this in mind; otherwise, they will face a memory issue, such as "running out of heap memory".
- it is very dangerous to run out of heap memory, as this will cause ESP32 an "undefined behavior". Thus, enough heap memory should be reserved for the application, so that it never runs out of it.
- the Wi-Fi throughput heavily depends on memory-related configurations, such as the TCP window size and Wi-Fi RX/TX dynamic buffer number.
- the peak heap memory that the ESP32 LwIP/Wi-Fi may consume depends on a number of factors, such as the maximum TCP/UDP connections that the application may have.
- the total memory that the application requires is also an important factor when considering memory configuration.

Due to these reasons, there is not a good-for-all application configuration. Rather, it is recommended to consider memory configurations separately for every different application.

Dynamic vs. Static Buffer

The default type of buffer in Wi-Fi drivers is "dynamic". Most of the time the dynamic buffer can significantly save memory. However, it makes the application programming a little more difficult, because in this case the application needs to consider memory usage in Wi-Fi.

lwIP also allocates buffers at the TCP/IP layer, and this buffer allocation is also dynamic. See *lwIP documentation section about memory use and performance*.

Peak Wi-Fi Dynamic Buffer

The Wi-Fi driver supports several types of buffer (refer to *Wi-Fi Buffer Configure*). However, this section is about the usage of the dynamic Wi-Fi buffer only. The peak heap memory that Wi-Fi consumes is the **theoretically-maximum memory** that the Wi-Fi driver consumes. Generally, the peak memory depends on:

- the number of dynamic RX buffers that are configured: wifi_rx_dynamic_buf_num
- the number of dynamic TX buffers that are configured: wifi_tx_dynamic_buf_num
- the maximum packet size that the Wi-Fi driver can receive: wifi_rx_pkt_size_max
- the maximum packet size that the Wi-Fi driver can send: wifi_tx_pkt_size_max

So, the peak memory that the Wi-Fi driver consumes can be calculated with the following formula:

> wifi_dynamic_peek_memory = (wifi_rx_dynamic_buf_num * wifi_rx_pkt_size_max) + (wifi_tx_dynamic_buf_num * wifi_tx_pkt_size_max)

Generally, the dynamic TX long buffers and dynamic TX long long buffers can be ignored, because they are management frames which only have a small impact on the system.

4.31.32 How to Improve Wi-Fi Performance

The performance of ESP32 Wi-Fi is affected by many parameters, and there are mutual constraints between each parameter. A proper configuration cannot only improve performance, but also increase available memory for applications and improve stability.

This section briefly explains the operating mode of the Wi-Fi/LwIP protocol stack and the role of each parameter. It also gives several recommended configuration ranks to help choose the appropriate rank according to the usage scenario.

Protocol Stack Operation Mode

The ESP32 protocol stack is divided into four layers: Application, LwIP, Wi-Fi, and Hardware.

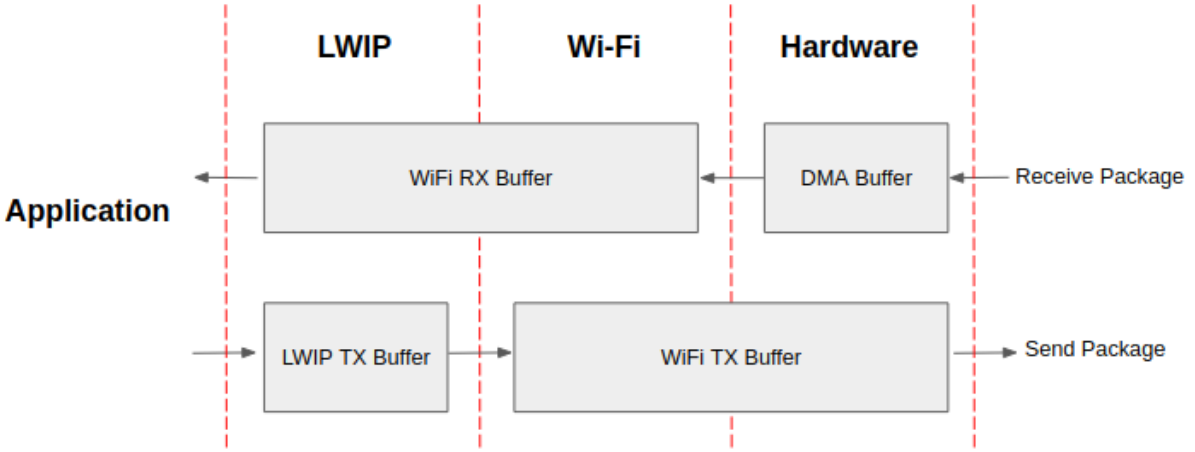


Fig. 62: ESP32 datapath

- During receiving, hardware puts the received packet into DMA buffer, and then transfers it into the RX buffer of Wi-Fi and LwIP in turn for related protocol processing, and finally to the application layer. The Wi-Fi RX buffer and the LwIP RX buffer shares the same buffer by default. In other words, the Wi-Fi forwards the packet to LwIP by reference by default.
- During sending, the application copies the messages to be sent into the TX buffer of the LwIP layer for TCP/IP encapsulation. The messages will then be passed to the TX buffer of the Wi-Fi layer for MAC encapsulation and wait to be sent.

Parameters

Increasing the size or number of the buffers mentioned above properly can improve Wi-Fi performance. Meanwhile, it will reduce available memory to the application. The following is an introduction to the parameters that users need to configure:

RX direction:

- ***CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM*** This parameter indicates the number of DMA buffer at the hardware layer. Increasing this parameter will increase the sender ' s one-time receiving throughput, thereby improving the Wi-Fi protocol stack ability to handle burst traffic.
- ***CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*** This parameter indicates the number of RX buffer in the Wi-Fi layer. Increasing this parameter will improve the performance of packet reception. This parameter needs to match the RX buffer size of the LwIP layer.
- ***CONFIG_ESP32_WIFI_RX_BA_WIN*** This parameter indicates the size of the AMPDU BA Window at the receiving end. This parameter should be configured to the smaller value between twice of *CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM* and *CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*.
- ***CONFIG_LWIP_TCP_WND_DEFAULT*** This parameter represents the RX buffer size of the LwIP layer for each TCP stream. Its value should be configured to the value of WIFI_DYNAMIC_RX_BUFFER_NUM (KB) to reach a high and stable performance. Meanwhile, in case of multiple streams, this value needs to be reduced proportionally.

TX direction:

- ***CONFIG_ESP32_WIFI_TX_BUFFER*** This parameter indicates the type of TX buffer, it is recommended to configure it as a dynamic buffer, which can make full use of memory.
- ***CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM*** This parameter indicates the number of TX buffer on the Wi-Fi layer. Increasing this parameter will improve the performance of packet sending. The parameter value needs to match the TX buffer size of the LwIP layer.
- ***CONFIG_LWIP_TCP_SND_BUF_DEFAULT*** This parameter represents the TX buffer size of the LwIP layer for each TCP stream. Its value should be configured to the value of

WIFI_DYNAMIC_TX_BUFFER_NUM (KB) to reach a high and stable performance. In case of multiple streams, this value needs to be reduced proportionally.

Throughput optimization by placing code in IRAM:

- *CONFIG_ESP32_WIFI_IRAM_OPT* If this option is enabled, some Wi-Fi functions are moved to IRAM, improving throughput. This increases IRAM usage by 15 kB.
- *CONFIG_ESP32_WIFI_RX_IRAM_OPT* If this option is enabled, some Wi-Fi RX functions are moved to IRAM, improving throughput. This increases IRAM usage by 16 kB.
- *CONFIG_LWIP_IRAM_OPTIMIZATION* If this option is enabled, some LwIP functions are moved to IRAM, improving throughput. This increases IRAM usage by 13 kB.

Note: The buffer size mentioned above is fixed as 1.6 KB.

How to Configure Parameters

The memory of ESP32 is shared by protocol stack and applications.

Here, several configuration ranks are given. In most cases, the user should select a suitable rank for parameter configuration according to the size of the memory occupied by the application.

The parameters not mentioned in the following table should be set to the default.

Rank	Iperf	TX prior	High-performance	RX prior	De-fault	Memory saving	Mini-mum
Available mem-ory (KB)	37.1	113.8	123.3	145.5	144.5	170.2	185.2
WIFI_STATIC_RX_BUFFER_NUM	16	6	6	6	6	6	4
WIFI_DYNAMIC_RX_BUFFER_NUM	64	16	24	34	20	12	8
WIFI_DYNAMIC_TX_BUFFER_NUM	64	28	24	18	20	12	8
WIFI_RX_BA_WIN	32	8	12	12	10	6	Dis-able
TCP_SND_BUF_DEFAULT (KB)	65	28	24	18	20	12	8
TCP_WND_DEFAULT (KB)	65	16	24	34	20	12	8
WIFI_IRAM_OPT	15	15	15	15	15	15	15
WIFI_RX_IRAM_OPT	16	16	16	16	16	16	16
LWIP_IRAM_OPTIMIZATION	13	13	13	13	13	13	13
TCP TX through-put (Mbit/s)	74.6	50.8	46.5	39.9	44.2	33.8	25.6
TCP RX through-put (Mbit/s)	63.6	35.5	42.3	48.5	40.5	30.1	27.8
UDP TX throughput (Mbit/s)	76.2	75.1	74.1	72.4	69.6	64.1	36.5
UDP RX throughput (Mbit/s)	83.1	66.3	75.1	75.6	73.1	65.3	54.7

Note: The test was performed with a single stream in a shielded box using an ASUS RT-N66U router. ESP32's CPU is dual core with 240 MHz. ESP32's flash is in QIO mode with 80 MHz.

Ranks:

- **Iperf rank** ESP32 extreme performance rank used to test extreme performance.
- **High-performance rank** The ESP32's high-performance configuration rank, suitable for scenarios where the application occupies less memory and has high-performance requirements. In this rank, users can choose to use the RX prior rank or the TX prior rank according to the usage scenario.
- **Default rank** ESP32's default configuration rank, the available memory, and performance are in balance.
- **Memory saving rank** This rank is suitable for scenarios where the application requires a large amount of memory, and the transceiver performance will be reduced in this rank.
- **Minimum rank** This is the minimum configuration rank of ESP32. The protocol stack only uses the necessary memory for running. It is suitable for scenarios where there is no requirement for performance and the application requires lots of space.

Using PSRAM

PSRAM is generally used when the application takes up a lot of memory. In this mode, the *CONFIG_ESP32_WIFI_TX_BUFFER* is forced to be static. *CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM* indicates the number of DMA buffers at the hardware layer, and increasing this parameter can improve performance. The following are the recommended ranks for using PSRAM:

Rank	Iperf	Default	Memory saving	Minimum
Available memory (KB)	113.8	152.4	181.2	202.6
WIFI_STATIC_RX_BUFFER_NUM	16	8	4	2
WIFI_DYNAMIC_RX_BUFFER_NUM	128	128	128	128
WIFI_STATIC_TX_BUFFER_NUM	16	8	4	2
WIFI_RX_BA_WIN	16	16	8	Disable
TCP_SND_BUF_DEFAULT (KB)	65	65	65	65
TCP_WND_DEFAULT (KB)	65	65	65	65
WIFI_IRAM_OPT	15	15	15	0
WIFI_RX_IRAM_OPT	16	16	0	0
LWIP_IRAM_OPTIMIZATION	0	0	0	0
TCP TX throughput (Mbit/s)	37.5	31.7	21.7	14.6
TCP RX throughput (Mbit/s)	31.5	29.8	26.5	21.1
UDP TX throughput (Mbit/s)	69.1	31.5	27.1	24.1
UDP RX throughput (Mbit/s)	40.1	38.5	37.5	36.9

4.31.33 Wi-Fi Menuconfig

Wi-Fi Buffer Configure

If you are going to modify the default number or type of buffer, it would be helpful to also have an overview of how the buffer is allocated/freed in the data path. The following diagram shows this process in the TX direction:

Description:

- The application allocates the data which needs to be sent out.
- The application calls TCPIP-/Socket-related APIs to send the user data. These APIs will allocate a PBUF used in LwIP, and make a copy of the user data.
- When LwIP calls a Wi-Fi API to send the PBUF, the Wi-Fi API will allocate a "Dynamic Tx Buffer" or "Static Tx Buffer", make a copy of the LwIP PBUF, and finally send the data.

The following diagram shows how buffer is allocated/freed in the RX direction:

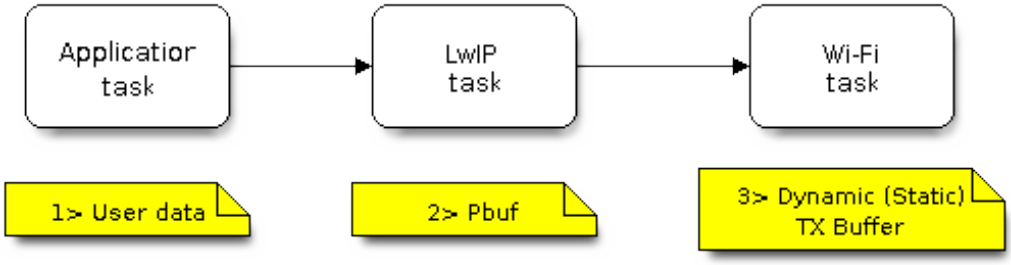


Fig. 63: TX Buffer Allocation

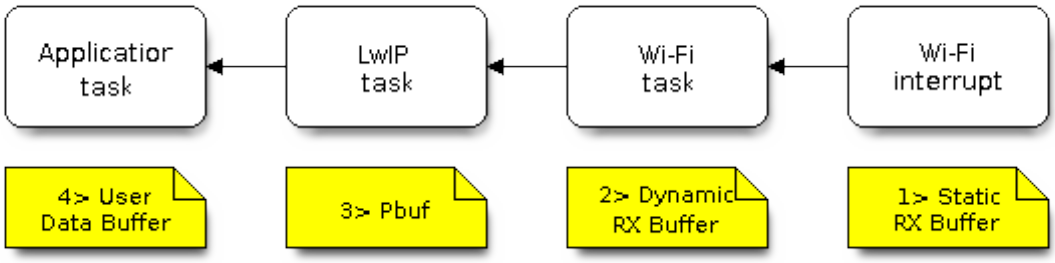


Fig. 64: RX Buffer Allocation

Description:

- The Wi-Fi hardware receives a packet over the air and puts the packet content to the "Static Rx Buffer", which is also called "RX DMA Buffer".
- The Wi-Fi driver allocates a "Dynamic Rx Buffer", makes a copy of the "Static Rx Buffer", and returns the "Static Rx Buffer" to hardware.
- The Wi-Fi driver delivers the packet to the upper-layer (LwIP), and allocates a PBUF for holding the "Dynamic Rx Buffer".
- The application receives data from LwIP.

The diagram shows the configuration of the Wi-Fi internal buffer.

Buffer Type	Alloc Type	Default	Configurable	Description
Static RX Buffer (Hardware RX Buffer)	Static	10 * 1600 Bytes	Yes	This is a kind of DMA memory. It is initialized in `esp_wifi_init()` and freed in `esp_wifi_deinit()`. The 'Static Rx Buffer' forms the hardware receiving list. Upon receiving a frame over the air, hardware writes the frame into the buffer and raises an interrupt to the CPU. Then, the Wi-Fi driver reads the content from the buffer and returns the buffer back to the list. If needs be, the application can reduce the memory statically allocated by Wi-Fi. It can reduce this value from 10 to 6 to save 6400 Bytes of memory. It is not recommended to reduce the configuration to a value less than 6 unless the AMPDU feature is disabled.
Dynamic RX Buffer	Dynamic	32	Yes	The buffer length is variable and it depends on the received frames' length. When the Wi-Fi driver receives a frame from the 'Hardware Rx Buffer', the 'Dynamic Rx Buffer' needs to be allocated from the heap. The number of the Dynamic Rx Buffer, configured in the menuconfig, is used to limit the total un-freed Dynamic Rx Buffer number.
Dynamic TX Buffer	Dynamic	32	Yes	This is a kind of DMA memory. It is allocated to the heap. When the upper-layer (LwIP) sends packets to the Wi-Fi driver, it firstly allocates a 'Dynamic TX Buffer' and makes a copy of the upper-layer buffer. The Dynamic and Static TX Buffers are mutually exclusive.
Static TX Buffer	Static	16 * 1600Bytes	Yes	This is a kind of DMA memory. It is initialized in `esp_wifi_init()` and freed in `esp_wifi_deinit()`. When the upper-layer (LwIP) sends packets to the Wi-Fi driver, it firstly allocates a 'Static TX Buffer' and makes a copy of the upper-layer buffer. The Dynamic and Static TX Buffer are mutually exclusive. The TX buffer must be a DMA buffer. For this reason, if PSRAM is enabled, the TX buffer must be static.
Management Short Buffer	Dynamic	8	NO	Wi-Fi driver's internal buffer.
Management Long Buffer	Dynamic	32	NO	Wi-Fi driver's internal buffer.
Management Long Long Buffer	Dynamic	32	NO	Wi-Fi driver's internal buffer.

Wi-Fi NVS Flash

If the Wi-Fi NVS flash is enabled, all Wi-Fi configurations set via the Wi-Fi APIs will be stored into flash, and the Wi-Fi driver will start up with these configurations the next time it powers on/reboots. However, the application can choose to disable the Wi-Fi NVS flash if it does not need to store the configurations into persistent memory, or has its own persistent storage, or simply due to debugging reasons, etc.

Wi-Fi Aggregate MAC Protocol Data Unit (AMPDU)

ESP32 supports both receiving and transmitting AMPDU, and the AMPDU can greatly improve the Wi-Fi throughput.

Generally, the AMPDU should be enabled. Disabling AMPDU is usually for debugging purposes.

4.31.34 Troubleshooting

Please refer to a separate document with *Espressif Wireshark User Guide*.

Espressif Wireshark User Guide

1. Overview

1.1 What is Wireshark? Wireshark (originally named "Ethereal") is a network packet analyzer that captures network packets and displays the packet data as detailed as possible. It uses WinPcap as its interface to directly capture network traffic going through a network interface controller (NIC).

You could think of a network packet analyzer as a measuring device used to examine what is going on inside a network cable, just like a voltmeter is used by an electrician to examine what is going on inside an electric cable.

In the past, such tools were either very expensive, proprietary, or both. However, with the advent of Wireshark, all that has changed.

Wireshark is released under the terms of the GNU General Public License, which means you can use the software and the source code free of charge. It also allows you to modify and customize the source code.

Wireshark is, perhaps, one of the best open source packet analyzers available today.

1.2 Some Intended Purposes Here are some examples of how Wireshark is typically used:

- Network administrators use it to troubleshoot network problems.
- Network security engineers use it to examine security problems.
- Developers use it to debug protocol implementations.
- People use it to learn more about network protocol internals.

Beside these examples, Wireshark can be used for many other purposes.

1.3 Features The main features of Wireshark are as follows:

- Available for UNIX and Windows
- Captures live packet data from a network interface
- Displays packets along with detailed protocol information
- Opens/saves the captured packet data
- Imports/exports packets into a number of file formats, supported by other capture programs
- Advanced packet filtering
- Searches for packets based on multiple criteria

- Colorizes packets according to display filters
- Calculates statistics
- ···and a lot more!

1.4 Wireshark Can or Can't Do

- **Live capture from different network media**.
  Wireshark can capture traffic from different network media, including wireless LAN.
- **Import files from many other capture programs**.
  Wireshark can import data from a large number of file formats, supported by other capture programs.
- **Export files for many other capture programs**.
  Wireshark can export data into a large number of file formats, supported by other capture programs.
- **Numerous protocol dissectors**.
  Wireshark can dissect, or decode, a large number of protocols.
- **Wireshark is not an intrusion detection system**.
  It will not warn you if there are any suspicious activities on your network. However, if strange things happen, Wireshark might help you figure out what is really going on.
- **Wireshark does not manipulate processes on the network, it can only perform "measurements" within it**.
  Wireshark does not send packets on the network or influence it in any other way, except for resolving names (converting numerical address values into a human readable format), but even that can be disabled.

2. Where to Get Wireshark You can get Wireshark from the official website: https://www.wireshark.org/download.html

Wireshark can run on various operating systems. Please download the correct version according to the operating system you are using.

3. Step-by-step Guide **This demonstration uses Wireshark 2.2.6 on Linux.**

a) Start Wireshark

On Linux, you can run the shell script provided below. It starts Wireshark, then configures NIC and the channel for packet capture.

```
ifconfig $1 down
iwconfig $1 mode monitor
iwconfig $1 channel $2
ifconfig $1 up
Wireshark&
```

In the above script, the parameter `$1` represents NIC and `$2` represents channel. For example, `wlan0` in `./xxx.sh wlan0 6`, specifies the NIC for packet capture, and `6` identifies the channel of an AP or Soft-AP.

b) Run the Shell Script to Open Wireshark and Display Capture Interface

c) Select the Interface to Start Packet Capture

As the red markup shows in the picture above, many interfaces are available. The first one is a local NIC and the second one is a wireless NIC.

Please select the NIC according to your requirements. This document will use the wireless NIC to demonstrate packet capture.

Double click *wlan0* to start packet capture.

d) Set up Filters

Since all packets in the channel will be captured, and many of them are not needed, you have to set up filters to get the packets that you need.

Please find the picture below with the red markup, indicating where the filters should be set up.

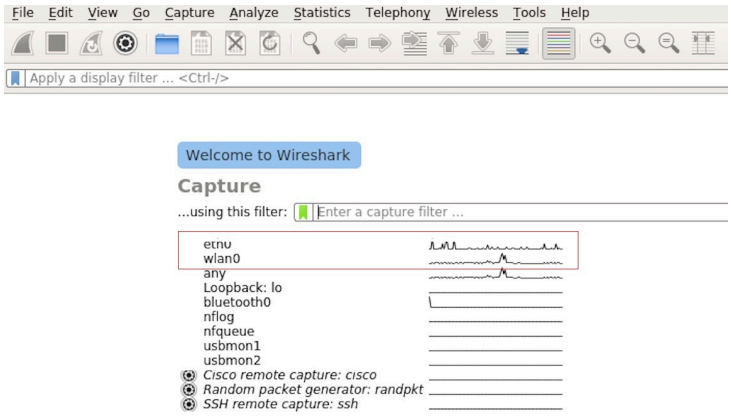


Fig. 65: Wireshark Capture Interface

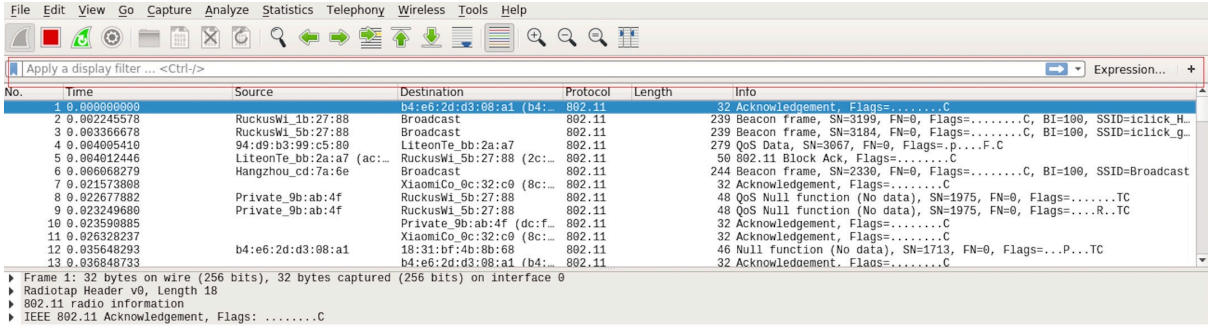


Fig. 66: Setting up Filters in Wireshark

Click *Filter*, the top left blue button in the picture below. The *display filter* dialogue box will appear.

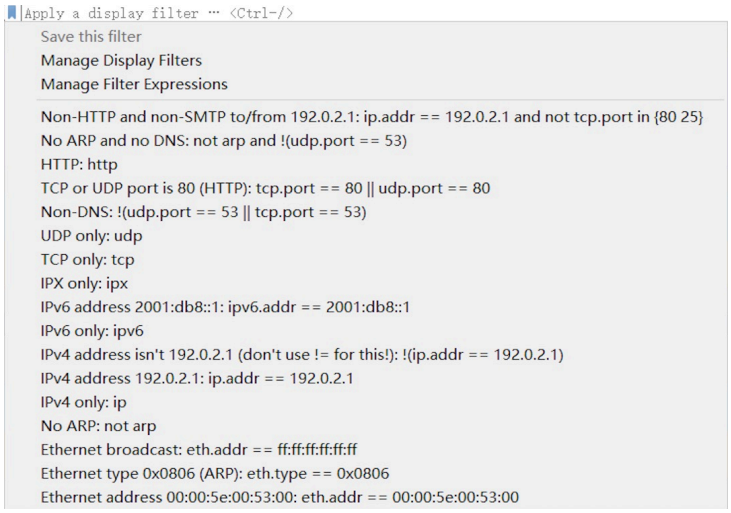


Fig. 67: *Display Filter* Dialogue Box

Click the *Expression* button to bring up the *Filter Expression* dialogue box and set the filter according to your requirements.

The quickest way: enter the filters directly in the toolbar.

Click on this area to enter or modify the filters. If you enter a wrong or unfinished filter, the built-in syntax check turns the background red. As soon as the correct expression is entered, the background becomes green.

The previously entered filters are automatically saved. You can access them anytime by opening the drop down list.

For example, as shown in the picture below, enter two MAC addresses as the filters and click *Apply* (the blue arrow). In this case, only the packet data transmitted between these two MAC addresses will be captured.

e) Packet List

You can click any packet in the packet list and check the detailed information about it in the box below the list. For example, if you click the first packet, its details will appear in that box.

f) Stop/Start Packet Capture

As shown in the picture below, click the red button to stop capturing the current packet.

Click the top left blue button to start or resume packet capture.

g) Save the Current Packet

On Linux, go to *File -> Export Packet Dissections -> As Plain Text File* to save the packet.

Please note that *All packets*, *Displayed* and *All expanded* must be selected.

By default, Wireshark saves the captured packet in a libpcap file. You can also save the file in other formats, e.g. txt, to analyze it in other tools.

4.32 Wi-Fi Security

4.32.1 ESP32 Wi-Fi Security Features

- Support for Protected Management Frames (PMF)
- Support for WPA3-Personal
- Support for Opportunistic Wireless Encryption

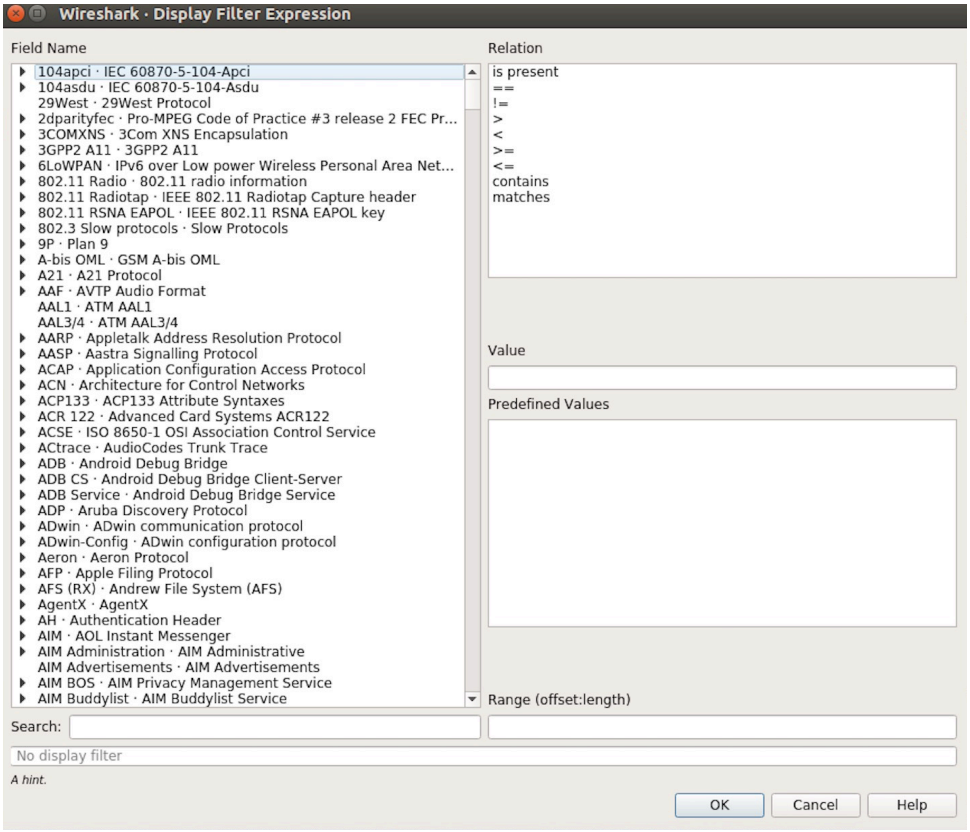


Fig. 68: *Filter Expression* Dialogue Box

Fig. 69: Filter Toolbar

Fig. 70: Example of MAC Addresses applied in the Filter Toolbar

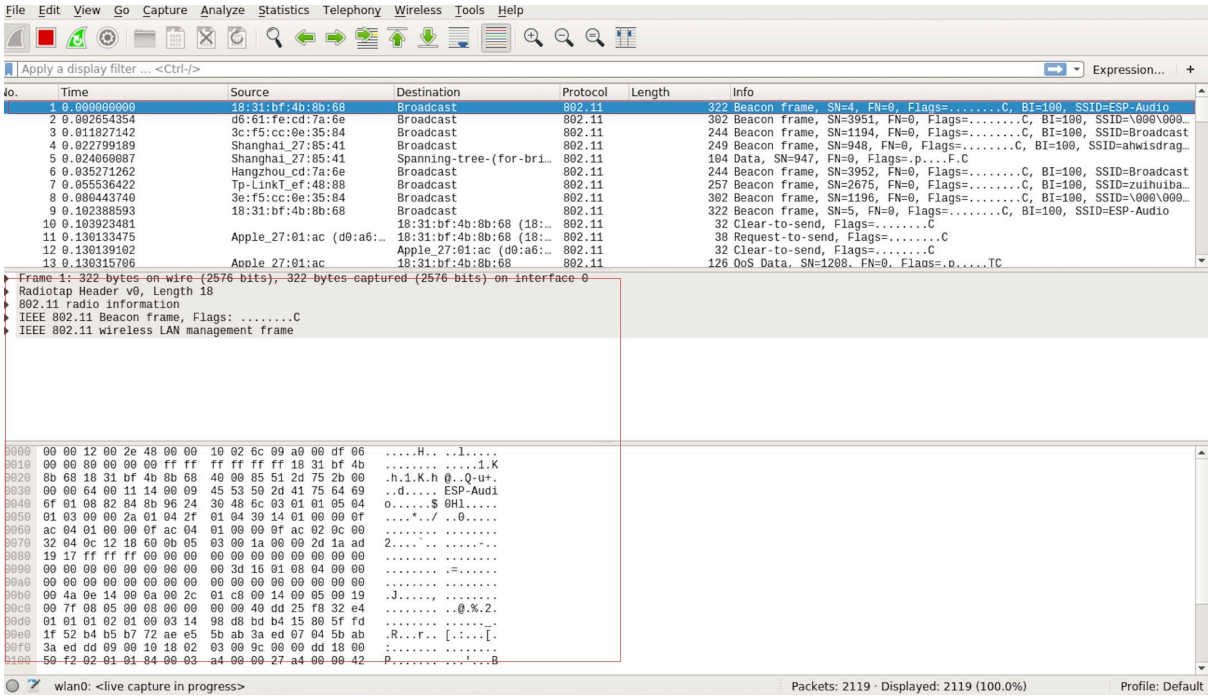


Fig. 71: Example of Packet List Details

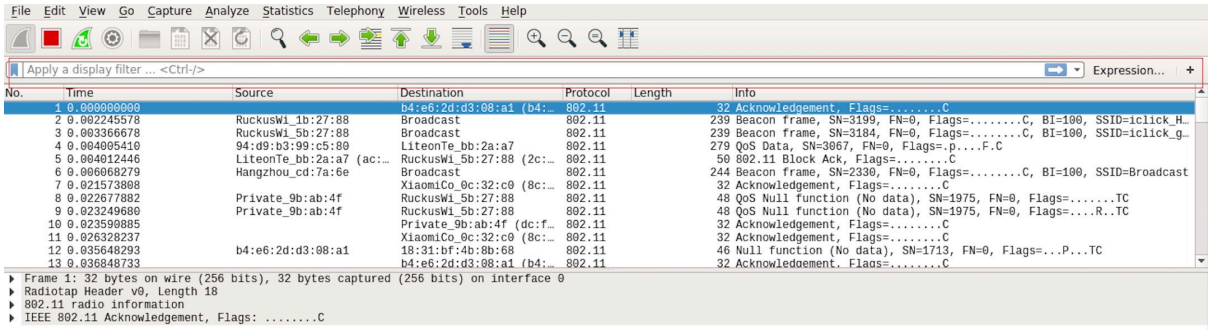


Fig. 72: Stopping Packet Capture

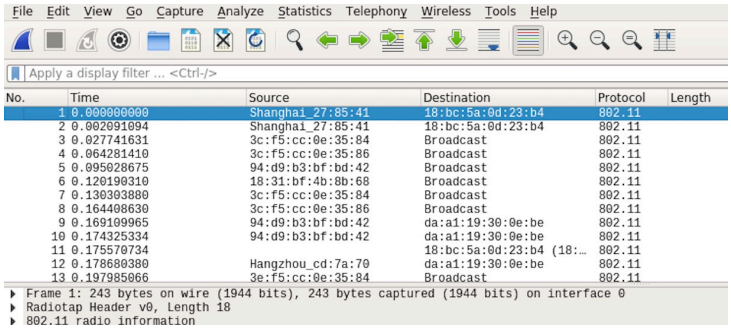


Fig. 73: Starting or Resuming the Packets Capture

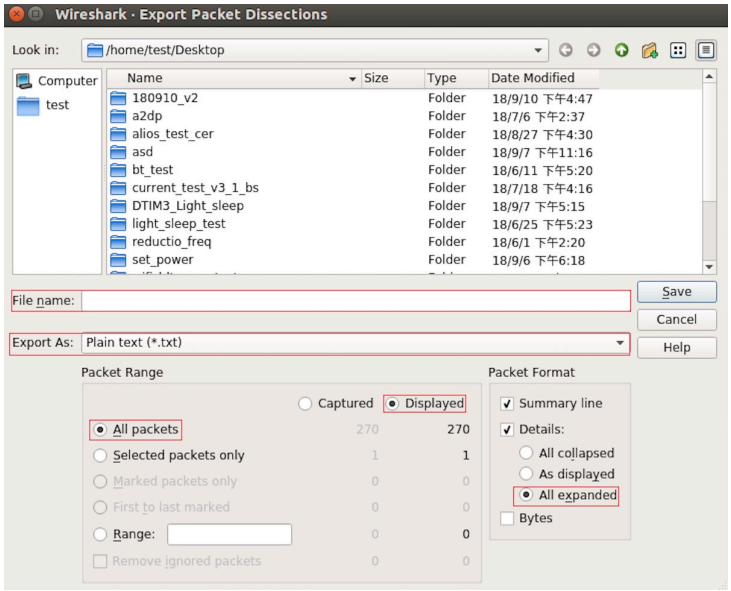


Fig. 74: Saving Captured Packets

In addition to traditional security methods (WEP/WPA-TKIP/WPA2-CCMP), ESP32 Wi-Fi supports state-of-the-art security protocols, namely Protected Management, Wi-Fi Protected Access 3 and Enhanced Open based on Opportunistic Wireless Encryption. WPA3 provides better privacy and robustness against known attacks on traditional modes. Enhanced Open enhances security and privacy of users connecting to open (public) Wireless Network without authentication.

4.32.2 Protected Management Frames (PMF)

Introduction

In Wi-Fi, management frames such as beacons, probes, (de)authentication, (dis)association are used by non-AP stations to scan and connect to an AP. Unlike data frames, these frames are sent unencrypted. An attacker can use eavesdropping and packet injection to send spoofed (de)authentication/(dis)association frames at the right time, leading to attacks such as Denial-of-Service (DOS) and man-in-the-middle

PMF provides protection against these attacks by encrypting unicast management frames and providing integrity checks for broadcast management frames. These include deauthentication, disassociation and robust management frames. It also provides Secure Association (SA) teardown mechanism to prevent spoofed association/authentication frames from disconnecting already connected clients.

There are 3 types of PMF configuration modes on both station and AP side -

- PMF Optional
- PMF Required
- PMF Disabled

API & Usage

ESP32 supports PMF in both Station and SoftAP mode. For both, the default mode is PMF Optional and disabling PMF is not possible. For even higher security, PMF required mode can be enabled by setting the `required` flag in *pmf_cfg* while using the `esp_wifi_set_config()` API. This will result in the device only connecting to a PMF enabled device and rejecting others.

> **Attention:** `capable` flag in *pmf_cfg* is deprecated and set to true internally. This is to take the additional security benefit of PMF whenever possible.

4.32.3 WiFi Enterprise

Introduction

Enterprise security is the secure authentication mechanism for enterprise wireless networks. It uses RADIUS server for authentication of network users before connecting to the Access Point. The authentication process is based on 802.1X policy and comes with different Extended Authentication Protocol (EAP) methods such as TLS, TTLS, PEAP and EAP-FAST. RADIUS server authenticates the users based on their credentials (username and password), digital certificates or both.

ESP32 supports WiFi Enterprise only in station mode.

ESP32 Supports **WPA2-Enterprise** and **WPA3-Enterprise**. WPA3-Enterprise builds upon the foundation of WPA2-Enterprise with the additional requirement of using Protected Management Frames (PMF) and server certificate validation on all WPA3 connections. **WPA3-Enterprise also offers an addition secure mode using 192-bit minimum-strength security protocols and cryptographic tools to better protect sensitive data.** The 192-bit security mode offered by WPA3-Enterprise ensures the right combination of cryptographic tools are used and sets a consistent baseline of security within a WPA3 network. WPA3-Enterprise 192-bit mode is only supported by modules having *SOC_WIFI_GCMP_SUPPORT* support. Enable *CONFIG_WPA_SUITE_B_192* flag to support WPA3-Enterprise with 192-bit mode.

ESP32 supports the following EAP methods:

- EAP-TLS: This is a certificate-based method and only requires SSID and EAP-IDF.
- PEAP: This is a Protected EAP method. Username and password are mandatory.
- **EAP-TTLS: This is a credential-based method. Only server authentication is mandatory while user authentication**
  - PAP: Password Authentication Protocol.
  - CHAP: Challenge Handshake Authentication Protocol.
  - MSCHAP and MSCHAP-V2.
- EAP-FAST: This is an authentication method based on Protected Access Credentials (PAC) which also uses identity and password. Currently, *CONFIG_WPA_MBEDTLS_TLS_CLIENT* flag should be disabled to use this feature.

Example wifi/wifi_enterprise demonstrates all the supported WiFi Enterprise methods except EAP-FAST. Please refer wifi/wifi_eap_fast for EAP-FAST example. EAP method can be selected from the Example Configuration menu in `idf.py menuconfig`. Refer to examples/wifi/wifi_enterprise/README.md for information on how to generate certificates and run the example.

4.32.4 WPA3-Personal

Introduction

Wi-Fi Protected Access-3 (WPA3) is a set of enhancements to Wi-Fi access security intended to replace the current WPA2 standard. It includes new features and capabilities that offer significantly better protection against different types of attacks. It improves upon WPA2-Personal in following ways:

- WPA3 uses Simultaneous Authentication of Equals (SAE), which is password-authenticated key agreement method based on Diffie-Hellman key exchange. Unlike WPA2, the technology is resistant to offline-dictionary attack, where the attacker attempts to determine shared password based on captured 4-way handshake without any further network interaction.
- Disallows outdated protocols such as TKIP, which is susceptible to simple attacks like MIC key recovery attack.
- Mandates Protected Management Frames (PMF), which provides protection for unicast and multicast robust management frames which include Disassoc and Deauth frames. This means that the attacker cannot disrupt an established WPA3 session by sending forged Assoc frames to the AP or Deauth/Disassoc frames to the Station.
- Provides forward secrecy, which means the captured data cannot be decrypted even if password is compromised after data transmission.

ESP32 station also supports following additional Wi-Fi CERTIFIED WPA3™ features.

- **Transition Disable** : WPA3 defines transition modes for client devices so that they can connect to a network even when some of the APs in that network do not support the strongest security mode. Client device implementations typically configure network profiles in a transition mode by default. However, such a client device could be subject to an active downgrade attack in which the attacker causes the client device to use a lower security mode in order to exploit a vulnerability with that mode. WPA3 has introduced the Transition Disable feature to mitigate such attacks, by enabling client devices to change from a transition mode to an "only" mode when connecting to a network, once that network indicates it fully supports the higher security mode. Enable `transition_disable` in *`wifi_sta_config_t`* to enable this feature for ESP32 station.
- **SAE PWE Methods**: ESP32 station supports SAE Password Element derivation method *Hunting And Pecking* and *Hash to Element (H2E)*. H2E is computationally efficient as it uses less iterations than Hunt and Peck, also it mitigates side channel attacks. These can be configured using parameter `sae_pwe_h2e` from *`wifi_sta_config_t`* for station. Hunt and peck, H2E both can be enabled by using *`WPA3_SAE_PWE_BOTH`* configuration.

Please refer to Security section of Wi-Fi Alliance's official website for further details.

Setting up WPA3 Personal with ESP32

A config option *CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE* is provided to Enable/Disable WPA3 for station. By default it is kept enabled, if disabled ESP32 will not be able to establish a WPA3 connection. Additionally, since PMF is mandated by WPA3 protocol, PMF Mode Optional is set by default for station. PMF Required can be configured using WiFi config.

Refer to *Protected Management Frames (PMF)* on how to set this mode.

After configuring all required settings for WPA3-Personal station, application developers need not worry about the underlying security mode of the AP. WPA3-Personal is now the highest supported protocol in terms of security, so it will be automatically selected for the connection whenever available. For example, if an AP is configured to be in WPA3 Transition Mode, where it will advertise as both WPA2 and WPA3 capable, Station will choose WPA3 for the connection with above settings. Note that Wi-Fi stack size requirement will increase 3kB when "Enable WPA3-Personal" is used.

4.32.5 Wi-Fi Enhanced Open™

Introduction

Enhanced open is used for providing security and privacy to users connecting to open (public) wireless networks, particularly in scenarios where user authentication is not desired or distribution of credentials impractical. Each user is provided with unique individual encryption keys that protect data exchange between a user device and the Wi-Fi network. Protected Management Frames further protects management traffic between the access point and user device. Enhanced Open is based on Opportunistic Wireless Encryption (OWE) standard. OWE Transition Mode enables a seamless transition from Open unencrypted WLANs to OWE WLANs without adversely impacting the end user experience.

ESP32 supports Wi-Fi Enhanced Open™ only in station mode.

Setting up OWE with ESP32

A config option *CONFIG_ESP32_WIFI_ENABLE_WPA3_OWE_STA* and configuration parameter `owe_enabled` in *`wifi_sta_config_t`* is provided to enable OWE support for station. To use OWE transition mode, along with the config provided above, *authmode* from *`wifi_scan_threshold_t`* should be set to *WIFI_AUTH_OPEN*.

4.33 RF Coexistence

4.33.1 Overview

ESP32 has only one 2.4 GHz ISM band RF module, which is shared by Bluetooth (BT & BLE) and Wi-Fi, so Bluetooth can't receive or transmit data while Wi-Fi is receiving or transmitting data and vice versa. Under such circumstances, ESP32 uses the time-division multiplexing method to receive and transmit packets.

4.33.2 Supported Coexistence Scenario for ESP32

Table 32: Supported Features of Wi-Fi and BLE Coexistence

			BLE			
			Scan	Advertising	Connecting	Connected
Wi-Fi	STA	Scan	Y	Y	Y	Y
		Connecting	Y	Y	Y	Y
		Connected	Y	Y	Y	Y
	SOFTAP	TX Beacon	Y	Y	Y	Y
		Connecting	C1	C1	C1	C1
		Connected	C1	C1	C1	C1
	Sniffer	RX	C1	C1	C1	C1
	ESP-NOW	RX	S	S	S	S
		TX	Y	Y	Y	Y

Table 33: Supported Features of Wi-Fi and Classic Bluetooth (BT) Coexistence

			BR/EDR				
			Inquiry	Inquiry scan	Page	Page scan	Connected
Wi-Fi	STA	Scan	Y	Y	Y	Y	Y
		Connecting	Y	Y	Y	Y	Y
		Connected	Y	Y	Y	Y	Y
	SOFTAP	TX Beacon	Y	Y	Y	Y	Y
		Connecting	C1	C1	C1	C1	C1
		Connected	C1	C1	C1	C1	C1
	Sniffer	RX	C1	C1	C1	C1	C1
	ESP-NOW	RX	S	S	S	S	S
		TX	Y	Y	Y	Y	Y

Note: Y: supported and performance is stable C1: supported but the performance is unstable X: not supported S: supported and performance is stable in STA mode, otherwise not supported

4.33.3 Coexistence Mechanism and Policy

Coexistence Mechanism

The RF resource allocation mechanism is based on priority. As shown below, both Bluetooth module and Wi-Fi module request RF resources from the coexistence module, and the coexistence module decides who will use the RF resource based on their priority.

Coexistence Policy

Coexistence Period and Time Slice Wi-Fi, BT, and BLE have their fixed time slice to use the RF. A coexistence period is divided into 3 time slices in the order of Wi-Fi, BT, and BLE. In the Wi-Fi slice, Wi-Fi's request to the coexistence arbitration module will have higher priority. Similarly, BT/BLE can enjoy higher priority at their

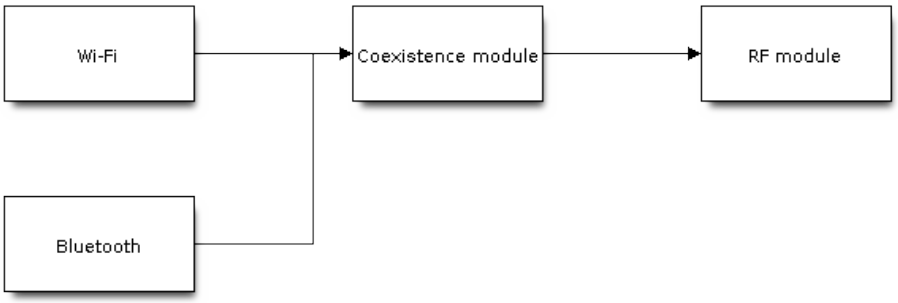


Fig. 75: Coexistence Mechanism

own time slices. The duration of the coexistence period and the proportion of each time slice are divided into four categories according to the Wi-Fi status:

1) IDLE status: the coexistence of BT and BLE is controlled by Bluetooth module.
2) CONNECTED status: the coexistence period starts at the Target Beacon Transmission Time (TBTT) and is more than 100 ms.
3) SCAN status: Wi-Fi slice and coexistence period are longer than in the CONNECTED status. To ensure Bluetooth performance, the Bluetooth time slice will also be adjusted accordingly.
4) CONNECTING status: Wi-Fi slice is longer than in the CONNECTED status. To ensure Bluetooth performance, the Bluetooth time slice will also be adjusted accordingly.

According to the coexistence logic, different coexistence periods and time slice strategies will be selected based on the Wi-Fi and Bluetooth usage scenarios. A Coexistence policy corresponding to a certain usage scenarios is called a "coexistence scheme". For example, the scenario of Wi-Fi CONNECTED and BLE CONNECTED has a corresponding coexistence scheme. In this scheme, the time slices of Wi-Fi and BLE in a coexistence period each account for 50%. The time allocation is shown in the following figure:

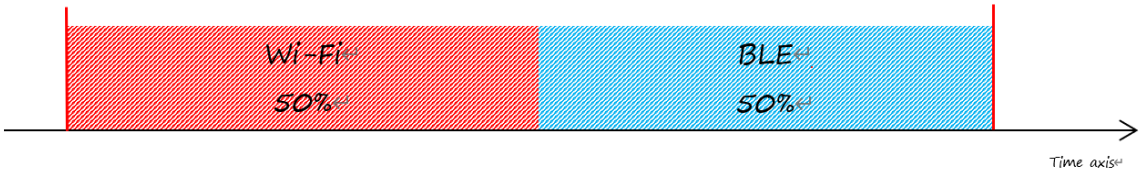


Fig. 76: Time Slice Under the Status of Wi-Fi CONNECTED and BLE CONNECTED

Dynamic Priority The coexistence module assigns different priorities to different status of Wi-Fi and Bluetooth. And the priority for each status is dynamic. For example, in every N BLE Advertising events, there is always one event with high priority. If a high-priority BLE Advertising event occurs within the Wi-Fi time slice, the right to use the RF may be preempted by BLE.

Wi-Fi Connectionless Modules Coexistence To some extent, some combinations of connectionless power-saving parameters *Window* and *Interval* would lead to extra Wi-Fi priority request out of Wi-Fi time slice. It`s for obtaining RF resources at coexistence for customized parameters, while leading to impact on Bluetooth performance.

If connectionless power-saving parameters are configured with default values, the coexistence module would perform in stable mode and the behaviour above would not happen. So please configure Wi-Fi connectionless power-saving parameters to default values unless you have plenty of coexistence performance tests for customized parameters.

Please refer to *connectionless module power save* to get more detail.

4.33.4 How to Use the Coexistence Feature

Coexistence API

For most coexistence cases, ESP32 will switch the coexistence status automatically without calling API. However, ESP32 provides two APIs for the coexistence of BLE MESH and Wi-Fi. When the status of BLE MESH changes, call `esp_coex_status_bit_clear` to clear the previous status first and then call `esp_coex_status_bit_set` to set the current status.

BLE MESH Coexistence Status As the firmware of Wi-Fi and Bluetooth are not aware of the current scenario of the upper layer application, some coexistence schemes require application code to call the coexistence API to take effect. The application layer needs to pass the working status of BLE MESH to the coexistence module for selecting the coexistence scheme.

- ESP_COEX_BLE_ST_MESH_CONFIG: network is provisioning
- ESP_COEX_BLE_ST_MESH_TRAFFIC: data is transmitting
- ESP_COEX_BLE_ST_MESH_STANDBY: in idle status with no significant data interaction

Coexistence API Error Codes

All coexistence APIs have custom return values, i.e. error codes. These error codes can be categorized as:

- No error. For example, the return value ESP_OK siginifies the API returned successfully.
- Recoverable errors. For example, the return value ESP_ERR_INVALID_ARG signifies API parameter errors.

Setting Coexistence Compile-time Options

- After writing the coexistence program, you must check *CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE* option through menuconfig to open coexistence configuration on software, otherwise the coexistence function mentioned above cannot be used.
- To ensure better communication performance of Wi-Fi and Bluetooth in the case of coexistence, run the task of the Wi-Fi protocol stack, the task of the Bluetooth Controller and Host protocol stack on different CPUs. You can use *CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE* and *CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE* (or *CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE*) to put the tasks of the Bluetooth controller and the host protocol stack on the same CPU, and then use *CONFIG_ESP32_WIFI_TASK_CORE_ID* to place the task of the Wi-Fi protocol stack on another CPU.
- In the case of coexistence, BLE SCAN may be interrupted by Wi-Fi and Wi-Fi releases RF resources before the end of the current BLE scan window. In order to make BLE acquire RF resources again within the current scan window, you can check the FULL SCAN configuration option through *CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED*.
- You can reduce the memory consumption by configuring the following options on menuconfig.
  - *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY*: enable the configuration of dynamic memory for Bluetooth protocol stack.
  - *CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM*: reduce the number of Wi-Fi static RX buffers.
  - *CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*: reduce the number of Wi-Fi dynamic RX buffers.
  - *CONFIG_ESP32_WIFI_TX_BUFFER*: enable the configuration of dynamic allocation TX buffers.
  - *CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM*: reduce the number of Wi-Fi dynamic TX buffers.
  - *CONFIG_ESP32_WIFI_TX_BA_WIN*: reduce the number of Wi-Fi Block Ack TX windows.
  - *CONFIG_ESP32_WIFI_RX_BA_WIN*: reduce the number of Wi-Fi Block Ack RX windows.

  - *CONFIG_ESP32_WIFI_MGMT_SBUF_NUM*: reduce the number of Wi-Fi Management Short Buffer.
  - *CONFIG_ESP32_WIFI_RX_IRAM_OPT*: turning off this configuration option will reduce the IRAM memory by approximately 17 KB.
  - *CONFIG_LWIP_TCP_SND_BUF_DEFAULT*: reduce the default TX buffer size for TCP sockets.
  - *CONFIG_LWIP_TCP_WND_DEFAULT*: reduce the default size of the RX window for TCP sockets.
  - *CONFIG_LWIP_TCP_RECVMBOX_SIZE*: reduce the size of the TCP receive mailbox. Receive mailbox buffers data within active connections and handles data flow during connections。
  - *CONFIG_LWIP_UDP_RECVMBOX_SIZE*: reduce the size of the UDP receive mailbox.
  - *CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*: reduce the size of TCPIP task receive mailbox.

Note: Since the coexistence configuration option depends on the Bluetooth configuration option, please turn on the Bluetooth configuration option first before configuring the coexistence feature in the Wi-Fi configuration option.

4.34 Reproducible Builds

4.34.1 Introduction

ESP-IDF build system has support for reproducible builds.

When reproducible builds are enabled, the application built with ESP-IDF doesn't depend on the build environment. Both the .elf file and .bin files of the application remains exactly the same, even if the following variables change:

- Directory where the project is located
- Directory where ESP-IDF is located (`IDF_PATH`)
- Build time

4.34.2 Reasons for non-reproducible builds

There are several reasons why an application may depend on the build environment, even when the same source code and tools versions are used.

- In C code, `__FILE__` preprocessor macro is expanded to the full path of the source file.
- `__DATE__` and `__TIME__` preprocessor macros are expanded to compilation date and time.
- When the compiler generates object files, it adds sections with debug information. These sections help debuggers, like GDB, to locate the source code which corresponds to a particular location in the machine code. These sections typically contain paths of relevant source files. These paths may be absolute, and will include the path to ESP-IDF or to the project.

There are also other possible reasons, such as unstable order of inputs and non-determinism in the build system.

4.34.3 Enabling reproducible builds in ESP-IDF

Reproducible builds can be enabled in ESP-IDF using *CONFIG_APP_REPRODUCIBLE_BUILD* option.

This option is disabled by default. It can be enabled in `menuconfig`.

The option may also be added into `sdkconfig.defaults`. If adding the option into `sdkconfig.defaults`, delete the `sdkconfig` file and run the build again. See *Custom sdkconfig defaults* for more information.

4.34.4 How reproducible builds are achieved

ESP-IDF achieves reproducible builds using the following measures:

- In ESP-IDF source code, `__DATE__` and `__TIME__` macros are not used when reproducible builds are enabled. Note, if the application source code uses these macros, the build will not be reproducible.

- ESP-IDF build system passes a set of `-fmacro-prefix-map` and `-fdebug-prefix-map` flags to replace base paths with placeholders:
  - Path to ESP-IDF is replaced with `/IDF`
  - Path to the project is replaced with `/IDF_PROJECT`
  - Path to the build directory is replaced with `/IDF_BUILD`
  - Paths to components are replaced with `/COMPONENT_NAME_DIR` (where `NAME` is the name of the component)
- Build date and time are not included into the *application metadata structure* if *CONFIG_APP_REPRODUCIBLE_BUILD* is enabled.
- ESP-IDF build system ensures that source file lists, component lists and other sequences are sorted before passing them to CMake. Various other parts of the build system, such as the linker script generator also perform sorting to ensure that same output is produced regardless of the environment.

4.34.5 Reproducible builds and debugging

When reproducible builds are enabled, file names included in debug information sections are altered as shown in the previous section. Due to this fact, the debugger (GDB) is not able to locate the source files for the given code location.

This issue can be solved using GDB `set substitute-path` command. For example, by adding the following command to GDB init script, the altered paths can be reverted to the original ones:

```
set substitute-path /COMPONENT_FREERTOS_DIR /home/user/esp/esp-idf/components/
↪freertos
```

ESP-IDF build system generates a file with the list of such `set substitute-path` commands automatically during the build process. The file is called `prefix_map_gdbinit` and is located in the project `build` directory.

When *idf.py gdb* is used to start debugging, this additional gdbinit file is automatically passed to GDB. When launching GDB manually or from and IDE, please pass this additional gdbinit script to GDB using `-x build/prefix_map_gdbinit` argument.

4.34.6 Factors which still affect reproducible builds

Note that the built application still depends on:

- ESP-IDF version
- Versions of the build tools (CMake, Ninja) and the cross-compiler

IDF Docker Image can be used to ensure that these factors do not affect the build.

4.35 Low Power Mode User Guide

The document has not been translated into English yet. In the meantime, please refer to the Chinese version.

Chapter 5

Migration Guides

5.1 ESP-IDF 5.x Migration Guide

5.1.1 Migration from 4.4 to 5.0

Bluetooth Classic

Bluedroid

- bt/host/bluedroid/api/include/api/esp_hf_defs.h
  - In *esp_hf_cme_err_t*
    - ESP_HF_CME_MEMEORY_FULL renamed to ESP_HF_CME_MEMORY_FULL
    - ESP_HF_CME_MEMEORY_FAILURE renamed to ESP_HF_CME_MEMORY_FAILURE
- bt/host/bluedroid/api/include/api/esp_hf_ag_api.h
  - esp_bt_hf_init(esp_bd_addr_t remote_addr) changes into esp_hf_ag_init(void).
  - esp_bt_hf_deinit(esp_bd_addr_t remote_addr) changes into esp_hf_ag_deinit(void)

    Along with this change, the *bt_bdaddr_t init* and *bt_bdaddr_t deinit* has been removed from *union btc_arg_t*.
  - esp_bt_hf_register_callback is now changed into esp_hf_ag_register_callback.
  - esp_bt_hf_connect changes into esp_hf_ag_slc_connect
  - esp_bt_hf_disconnect changes into esp_hf_ag_slc_disconnect
  - esp_bt_hf_connect_audio changes into esp_hf_ag_audio_connect
  - esp_bt_hf_disconnect_audio changes into esp_hf_ag_audio_disconnect
  - esp_bt_hf_vra changes into esp_hf_ag_vra_control
  - esp_bt_hf_volume_control changes into esp_hf_ag_volume_control
  - esp_hf_unat_response changes into esp_hf_ag_unknown_at_send
  - esp_bt_hf_cmee_response changes into esp_hf_ag_cmee_send
  - esp_bt_hf_indchange_notification changes into esp_hf_ag_devices_status_indchange
  - esp_bt_hf_cind_response changes into esp_hf_ag_cind_response
  - esp_bt_hf_cops_response changes into esp_hf_ag_cops_response
  - esp_bt_hf_clcc_response changes into esp_hf_ag_clcc_response
  - esp_bt_hf_cnum_response changes into esp_hf_ag_cnum_response
  - esp_bt_hf_bsir changes into esp_hf_ag_bsir
  - esp_bt_hf_answer_call changes into esp_hf_ag_answer_call
  - esp_bt_hf_reject_call changes into esp_hf_ag_reject_call

- `esp_bt_hf_out_call` changes into `esp_hf_ag_out_call`
- `esp_bt_hf_end_call` changes into `esp_hf_ag_end_call`
- `esp_bt_hf_register_data_callback` changes into `esp_hf_ag_register_data_callback`
- `esp_hf_outgoing_data_ready` changes into `esp_hf_ag_outgoing_data_ready`

Bluetooth Low Energy

Bluedroid

The following Bluedroid macros, types, and functions have been renamed:

- bt/host/bluedroid/api/include/api/esp_gap_ble_api.h
  - In *esp_gap_ble_cb_event_t*
    - `ESP_GAP_BLE_SET_PREFERED_DEFAULT_PHY_COMPLETE_EVT` renamed to `ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT`
    - `ESP_GAP_BLE_SET_PREFERED_PHY_COMPLETE_EVT` renamed to `ESP_GAP_BLE_SET_PREFERRED_PHY_COMPLETE_EVT`
    - `ESP_GAP_BLE_CHANNEL_SELETE_ALGORITHM_EVT` renamed to `ESP_GAP_BLE_CHANNEL_SELECT_ALGORITHM_EVT`
  - `esp_ble_wl_opration_t` renamed to *esp_ble_wl_operation_t*
  - `esp_ble_gap_cb_param_t.pkt_data_lenth_cmpl` renamed to `pkt_data_length_cmpl`
  - `esp_ble_gap_cb_param_t.update_whitelist_cmpl.wl_opration` renamed to `wl_operation`
  - `esp_ble_gap_set_prefered_default_phy` renamed to *esp_ble_gap_set_preferred_default_phy()*
  - `esp_ble_gap_set_prefered_phy` renamed to *esp_ble_gap_set_preferred_phy()*
- bt/host/bluedroid/api/include/api/esp_gatt_defs.h
  - In *esp_gatt_status_t*
    - `ESP_GATT_ENCRYPED_MITM` renamed to `ESP_GATT_ENCRYPTED_MITM`
    - `ESP_GATT_ENCRYPED_NO_MITM` renamed to `ESP_GATT_ENCRYPTED_NO_MITM`

Nimble

The following Nimble APIs have been removed:

- bt/host/nimble/esp-hci/include/esp_nimble_hci.h
  - Remove : esp_err_t esp_nimble_hci_and_controller_init(void);
    - Controller initialization, enable and HCI initialization calls have been moved to *nimble_port_init*. This function can be deleted directly.
  - Remove : esp_err_t esp_nimble_hci_and_controller_deinit(void);
    - Controller deinitialization, disable and HCI deinitialization calls have been moved to *nimble_port_deinit*. This function can be deleted directly.

ESP-BLE-MESH

The following ESP-BLE-MESH macro has been renamed:

- bt/esp_ble_mesh/api/esp_ble_mesh_defs.h
  - In *esp_ble_mesh_prov_cb_event_t*:
    - `ESP_BLE_MESH_PROVISIONER_DRIECT_ERASE_SETTINGS_COMP_EVT` renamed to `ESP_BLE_MESH_PROVISIONER_DIRECT_ERASE_SETTINGS_COMP_EVT`

Build System

Migrating from GNU Make Build System ESP-IDF v5.0 no longer supports GNU make-based projects. Please follow the *build system* guide for migration.

Update Fragment File Grammar Please follow the *migrate linker script fragment files grammar* chapter for migrating v3.x grammar to the new one.

Specify Component Requirements Explicitly In previous versions of ESP-IDF, some components were always added as public requirements (dependencies) to every component in the build, in addition to the *common component requirements*:

- `driver`
- `efuse`
- `esp_timer`
- `lwip`
- `vfs`
- `esp_wifi`
- `esp_event`
- `esp_netif`
- `esp_eth`
- `esp_phy`

This means that it was possible to include header files of those components without specifying them as requirements in `idf_component_register`. This behavior was caused by transitive dependencies of various common components.

In ESP-IDF v5.0, this behavior is fixed and these components are no longer added as public requirements by default.

Every component depending on one of the components which isn't part of common requirements has to declare this dependency explicitly. This can be done by adding `REQUIRES <component_name>` or `PRIV_REQUIRES <component_name>` in `idf_component_register` call inside component's `CMakeLists.txt`. See *Component Requirements* for more information on specifying requirements.

Setting `COMPONENT_DIRS` and `EXTRA_COMPONENT_DIRS` Variables ESP-IDF v5.0 includes a number of improvements to support building projects with space characters in their paths. To make that possible, there are some changes related to setting `COMPONENT_DIRS` and `EXTRA_COMPONENT_DIRS` variables in project CMakeLists.txt files.

Adding non-existent directories to `COMPONENT_DIRS` or `EXTRA_COMPONENT_DIRS` is no longer supported and will result in an error.

Using string concatenation to define `COMPONENT_DIRS` or `EXTRA_COMPONENT_DIRS` variables is now deprecated. These variables should be defined as CMake lists, instead. For example, use:

```
set(EXTRA_COMPONENT_DIRS path1 path2)
list(APPEND EXTRA_COMPONENT_DIRS path3)
```

instead of:

```
set(EXTRA_COMPONENT_DIRS "path1 path2")
set(EXTRA_COMPONENT_DIRS "${EXTRA_COMPONENT_DIRS} path3")
```

Defining these variables as CMake lists is compatible with previous ESP-IDF versions.

Update Usage of target_link_libraries with project_elf ESP-IDF v5.0 fixes CMake variable propagation issues for components. This issue caused compiler flags and definitions that were supposed to apply to one component to be applied to every component in the project.

As a side effect of this, user projects from ESP-IDF v5.0 onwards must use `target_link_libraries` with `project_elf` explicitly and custom CMake projects must specify `PRIVATE`, `PUBLIC`, or `INTERFACE` arguments. This is a breaking change and is not backward compatible with previous ESP-IDF versions.

For example:

```
target_link_libraries(${project_elf} PRIVATE "-Wl,--wrap=esp_panic_handler")
```

instead of:

```
target_link_libraries(${project_elf} "-Wl,--wrap=esp_panic_handler")
```

Update CMake Version In ESP-IDF v5.0 minimal CMake version was increased to 3.16 and versions lower than 3.16 are not supported anymore. Run `tools/idf_tools.py install cmake` to install a suitable version if your OS version doesn't have one.

This affects ESP-IDF users who use system-provided CMake and custom CMake.

Reorder the Applying of the Target-Specific Config Files ESP-IDF v5.0 reorders the applying order of target-specific config files and other files listed in SDKCONFIG_DEFAULTS. Now, target-specific files will be applied right after the file brings it in, before all latter files in `SDKCONFIG_DEFAULTS`.

For example:

```
If ``SDKCONFIG_DEFAULTS="sdkconfig.defaults;sdkconfig_devkit1"``, and there is a␣
↪file ``sdkconfig.defaults.esp32`` in the same folder, then the files will be␣
↪applied in the following order: (1) sdkconfig.defaults (2) sdkconfig.defaults.
↪esp32 (3) sdkconfig_devkit1.
```

If you have a key with different values in the target-specific files of the former item (e.g., `sdkconfig.defaults.esp32` above) and the latter item (e.g., `sdkconfig_devkit1` above), please note the latter will override the target-specific file of the former.

If you do want to have some target-specific config values, please put it into the target-specific file of the latter item (e.g., `sdkconfig_devkit1.esp32`).

GCC

GCC Version The previous GCC version was GCC 8.4.0. This has now been upgraded to GCC 11.2.0 on all targets. Users that need to port their code from GCC 8.4.0 to 11.2.0 should refer to the series of official GCC porting guides listed below:

- Porting to GCC 9
- Porting to GCC 10
- Porting to GCC 11

Warnings The upgrade to GCC 11.2.0 has resulted in the addition of new warnings, or enhancements to existing warnings. The full details of all GCC warnings can be found in GCC Warning Options. Users are advised to double-check their code, then fix the warnings if possible. Unfortunately, depending on the warning and the complexity of the user's code, some warnings will be false positives that require non-trivial fixes. In such cases, users can choose to suppress the warning in multiple ways. This section outlines some common warnings that users are likely to encounter, and ways to suppress them.

Warning: Users are advised to check that a warning is indeed a false positive before attempting to suppress them it.

-Wstringop-overflow, -Wstringop-overread, -Wstringop-truncation, and -Warray-bounds Users that use memory/string copy/compare functions will run into one of the `-Wstringop` warnings if the compiler cannot properly determine the size of the memory/string. The examples below demonstrate code that triggers these warnings and how to suppress them.

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wstringop-overflow"
#pragma GCC diagnostic ignored "-Warray-bounds"
    memset(RTC_SLOW_MEM, 0, CONFIG_ULP_COPROC_RESERVE_MEM); // <<-- This line␣
↪leads to warnings
#pragma GCC diagnostic pop
```

```
#pragma GCC diagnostic push
#if     __GNUC__ >= 11
#pragma GCC diagnostic ignored "-Wstringop-overread" // <<-- This key had been␣
↪introduced since GCC 11
#endif
#pragma GCC diagnostic ignored "-Warray-bounds"
            memcpy(backup_write_data, (void *)EFUSE_PGM_DATA0_REG, sizeof(backup_
↪write_data)); // <<-- This line leads to warnings
#pragma GCC diagnostic pop
```

-Waddress-of-packed-member GCC will issue this warning when accessing an unaligned member of a packed `struct` due to the incurred penalty of unaligned memory access. However, all ESP chips (on both Xtensa and RISC-V architectures) allow for unaligned memory access and incur no extra penalty. Thus, this warning can be ignored in most cases.

```
components/bt/host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c: In function␣
↪'btc_to_bta_gatt_id':
components/bt/host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c:105:21: warning:␣
↪taking address of packed member of 'struct <anonymous>' may result in an␣
↪unaligned pointer value [-Waddress-of-packed-member]
  105 |     btc_to_bta_uuid(&p_dest->uuid, &p_src->uuid);
      |                     ^~~~~~~~~~~~
```

If the warning occurs in multiple places across multiple source files, users can suppress the warning at the CMake level as demonstrated below.

```
set_source_files_properties(
    "host/bluedroid/bta/gatt/bta_gattc_act.c"
    "host/bluedroid/bta/gatt/bta_gattc_cache.c"
    "host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c"
    "host/bluedroid/btc/profile/std/gatt/btc_gatts.c"
    PROPERTIES COMPILE_FLAGS -Wno-address-of-packed-member)
```

However, if there are only one or two instances, users can suppress the warning directly in the source code itself as demonstrated below.

```
#pragma GCC diagnostic push
#if     __GNUC__ >= 9
#pragma GCC diagnostic ignored "-Waddress-of-packed-member" <<-- This key had been␣
↪introduced since GCC 9
#endif
    uint32_t* reg_ptr = (uint32_t*)src;
#pragma GCC diagnostic pop
```

`llabs()` for 64-bit Integers The function `abs()` from stdlib.h takes `int` argument. Please use `llabs()` for types that are intended to be 64-bit. It is particularly important for `time_t`.

Espressif Toolchain Changes

`int32_t` and `uint32_t` for Xtensa Compiler The types `int32_t` and `uint32_t` have been changed from the previous `int` and `unsigned int` to `long` and `unsigned long` respectively for the Xtensa compiler. This change now matches upstream GCC which `long` integers for `int32_t` and `uint32_t` on Xtensa, RISC-V, and other architectures.

	2021r2 and older, GCC 8	2022r1, GCC 11
Xtensa	(unsigned) int	(unsigned) long
riscv32	(unsigned) long	(unsigned) long

The change mostly affects code that formats strings using types provided by `<inttypes.h>`. Users will need to replace placeholders such as `%i` and `%x` with `PRIi32` and `PRIxx` respectively.

In other cases, it should be noted that enums have the `int` type.

In common, `int32_t` and `int`, as well as `uint32_t` and `unsigned int`, are different types.

Removing `CONFIG_COMPILER_DISABLE_GCC8_WARNINGS` Build Option `CONFIG_COMPILER_DISABLE_GCC8_WARNINGS` option was introduced to allow building of legacy code dating from the rigid GCC 5 toolchain. However, enough time has passed to allow for the warnings to be fixed, thus this option has been removed.

For now in GCC 11, users are advised to review their code and fix the compiler warnings where possible.

Networking

Wi-Fi

Callback function type esp_now_recv_cb_t Previously, the first parameter of `esp_now_recv_cb_t` was of type `const uint8_t *mac_addr`, which only included the address of ESP-NOW peer device.

This now changes. The first parameter is of type `esp_now_recv_info_t`, which has members `src_addr`, `des_addr` and `rx_ctrl`. Therefore, the following updates are required:

- Redefine ESP-NOW receive callback function.
- `src_addr` can be used to replace original `mac_addr`.
- `des_addr` is the destination MAC address of ESP-NOW packet, which can be unicast or broadcast address. With `des_addr`, the user can distinguish unicast and broadcast ESP-NOW packets where broadcast ESP-NOW packets can be non-encrypted even when encryption policy is configured for the ESP-NOW.
- `rx_ctrl` is Rx control info of the ESP-NOW packet, which provides more information about the packet.

Please refer to the ESP-NOW example: wifi/espnow/main/espnow_example_main.c

Ethernet

esp_eth_ioctl() API Previously, the *`esp_eth_ioctl()`* API had the following issues:

- The third parameter (which is of type `void *`) would accept an `int`/`bool` type arguments (i.e., not pointers) as input in some cases. However, these cases were not documented properly.
- To pass `int`/`bool` type argument as the third parameter, the argument had to be "unnaturally" casted to a `void *` type, to prevent a compiler warning as demonstrated in the code snippet below. This casting could lead to misuse of the *`esp_eth_ioctl()`* function.

```
esp_eth_ioctl(eth_handle, ETH_CMD_S_FLOW_CTRL, (void *)true);
```

Therefore, the usage of *esp_eth_ioctl()* is now unified. Arguments to the third parameter must be passed as pointers to a specific data type to/from where data will be stored/read by *esp_eth_ioctl()*. The code snippets below demonstrate the usage of *esp_eth_ioctl()*.

Usage example to set Ethernet configuration:

```
eth_duplex_t new_duplex_mode = ETH_DUPLEX_HALF;
esp_eth_ioctl(eth_handle, ETH_CMD_S_DUPLEX_MODE, &new_duplex_mode);
```

Usage example to get Ethernet configuration:

```
eth_duplex_t duplex_mode;
esp_eth_ioctl(eth_handle, ETH_CMD_G_DUPLEX_MODE, &duplex_mode);
```

KSZ8041/81 and LAN8720 Driver Update The KSZ8041/81 and LAN8720 drivers are updated to support more devices (i.e., generations) from their associated product families. The drivers can recognize particular chip numbers and their potential support by the driver.

As a result, the specific "chip number" functions calls are replaced by generic ones as follows:

- Removed `esp_eth_phy_new_ksz8041()` and `esp_eth_phy_new_ksz8081()`, and use *esp_eth_phy_new_ksz80xx()* instead
- Removed `esp_eth_phy_new_lan8720()`, and use *esp_eth_phy_new_lan87xx()* instead

ESP NETIF Glue Event Handlers `esp_eth_set_default_handlers()` and `esp_eth_clear_default_handlers()` functions are removed. Registration of the default IP layer handlers for Ethernet is now handled automatically. If you have already followed the suggestion to fully initialize the Ethernet driver and network interface before registering their Ethernet/IP event handlers, then no action is required (except for deleting the affected functions). Otherwise, you may start the Ethernet driver right after they register the user event handler.

PHY Address Auto-detect The Ethernet PHY address auto-detect function `esp_eth_detect_phy_addr()` is renamed to *esp_eth_phy_802_3_detect_phy_addr()* and its header declaration is moved to esp_eth/include/esp_eth_phy_802_3.h.

SPI-Ethernet Module Initialization The SPI-Ethernet Module initialization is now simplified. Previously, you had to manually allocate an SPI device using *spi_bus_add_device()* before instantiating the SPI-Ethernet MAC.

Now, you no longer need to call *spi_bus_add_device()* as SPI devices are allocated internally. As a result, the `eth_dm9051_config_t`, `eth_w5500_config_t`, and `eth_ksz8851snl_config_t` configuration structures are updated to include members for SPI device configuration (e.g., to allow fine tuning of SPI timing which may be dependent on PCB design). Likewise, the `ETH_DM9051_DEFAULT_CONFIG`, `ETH_W5500_DEFAULT_CONFIG`, and `ETH_KSZ8851SNL_DEFAULT_CONFIG` configuration initialization macros are updated to accept new input parameters. Refer to *Ethernet API Reference Guide* for an example of SPI-Ethernet Module initialization.

SPI-Ethernet Modules Initialization The SPI-Ethernet Module's initialization has been simplified. The previous initialization process required you to manually allocate an SPI device using *spi_bus_add_device()* before instantiating the SPI-Ethernet MAC.

Now, you no longer need to call *spi_bus_add_device()* as the allocation of the SPI device is done internally. As a result, the `eth_dm9051_config_t`, `eth_w5500_config_t`, and `eth_ksz8851snl_config_t` configuration structures were updated to include members for SPI device configuration (e.g., to allow fine tuning of SPI timing which may be dependent on PCB design). Likewise, the `ETH_DM9051_DEFAULT_CONFIG`,

`ETH_W5500_DEFAULT_CONFIG`, and `ETH_KSZ8851SNL_DEFAULT_CONFIG` configuration initialization macros have been updated to accept new input parameters. Refer to the *Ethernet API-Reference Guide* for an example of SPI-Ethernet Module initialization

TCP/IP Adapter The TCP/IP Adapter was a network interface abstraction component used in ESP-IDF prior to v4.1. This section outlines migration from tcpip_adapter API to its successor *ESP-NETIF*.

Updating Network Connection Code

Network Stack Initialization

- You may simply replace `tcpip_adapter_init()` with `esp_netif_init()`. However, please should note that the `esp_netif_init()` function now returns standard error codes. See *ESP-NETIF* for more details.
- The `esp_netif_deinit()` function is provided to de-initialize the network stack.
- You should also replace `#include "tcpip_adapter.h"` with `#include "esp_netif.h"`.

Network Interface Creation Previously, the TCP/IP Adapter defined the following network interfaces statically:

- WiFi Station
- WiFi Access Point
- Ethernet

This now changes. Network interface instance should be explicitly constructed, so that the *ESP-NETIF* can connect to the TCP/IP stack. For example, after the TCP/IP stack and the event loop are initialized, the initialization code for WiFi must explicitly call `esp_netif_create_default_wifi_sta();` or `esp_netif_create_default_wifi_ap();`.

Please refer to the example initialization code for these three interfaces:

- WiFi Station: wifi/getting_started/station/main/station_example_main.c
- WiFi Access Point: wifi/getting_started/softAP/main/softap_example_main.c
- Ethernet: ethernet/basic/main/ethernet_example_main.c

Other tcpip_adapter API Replacement All the tcpip_adapter functions have their esp-netif counter-part. Please refer to the esp_netif.h grouped into these sections:

- Setters/Getters
- DHCP
- DNS
- IP address

Default Event Handlers Event handlers are moved from tcpip_adapter to appropriate driver code. There is no change from application code perspective, as all events should be handled in the same way. Please note that for IP-related event handlers, application code usually receives IP addresses in the form of an esp-netif specific struct instead of the LwIP structs. However, both structs are binary compatible.

This is the preferred way to print the address:

```
ESP_LOGI(TAG, "got ip:" IPSTR "\n", IP2STR(&event->ip_info.ip));
```

Instead of

```
ESP_LOGI(TAG, "got ip:%s\n", ip4addr_ntoa(&event->ip_info.ip));
```

Since `ip4addr_ntoa()` is a LwIP API, the esp-netif provides `esp_ip4addr_ntoa()` as a replacement. However, the above method using `IP2STR()` is generally preferred.

IP Addresses You are advised to use esp-netif defined IP structures. Please note that with default compatibility enabled, the LwIP structs will still work.

- esp-netif IP address definitions

Next Steps To port an application which may fully benefit from the *ESP-NETIF*, you also need to disable the tcpip_adapter compatibility layer in the component configuration option. Please go to `ESP NETIF Adapter > Enable backward compatible tcpip_adapter interface`. After that, check if your project compiles.

The TCP/IP adapter includes many dependencies. Thus, disabling its compatibility might help separate the application from using specific TCP/IP stack API directly.

Peripherals

Peripheral Clock Gating As usual, peripheral clock gating is still handled by driver itself, users don't need to take care of the peripheral module clock gating.

However, for advanced users who implement their own drivers based on `hal` and `soc` components, the previous clock gating include path has been changed from `driver/periph_ctrl.h` to `esp_private/periph_ctrl.h`.

RTC Subsystem Control RTC control APIs have been moved from `driver/rtc_cntl.h` to `esp_private/rtc_ctrl.h`.

ADC

ADC Oneshot & Continuous Mode drivers The ADC oneshot mode driver has been redesigned.

- The new driver is in `esp_adc` component and the include path is `esp_adc/adc_oneshot.h`.
- The legacy driver is still available in the previous include path `driver/adc.h`.

The ADC continuous mode driver has been moved from `driver` component to `esp_adc` component.

- The include path has been changed from `driver/adc.h` to `esp_adc/adc_continuous.h`.

Attempting to use the legacy include path `driver/adc.h` of either driver will trigger the build warning below by default. However, the warning can be suppressed by enabling the *CONFIG_ADC_SUPPRESS_DEPRECATE_WARN* Kconfig option.

```
legacy adc driver is deprecated, please migrate to use esp_adc/adc_oneshot.h and␣
↪esp_adc/adc_continuous.h for oneshot mode and continuous mode drivers␣
↪respectively
```

ADC Calibration Driver The ADC calibration driver has been redesigned.

- The new driver is in `esp_adc` component and the include path is `esp_adc/adc_cali.h` and `esp_adc/adc_cali_scheme.h`.

Legacy driver is still available by including `esp_adc_cal.h`. However, if users still would like to use the include path of the legacy driver, users should add `esp_adc` component to the list of component requirements in CMakeLists.txt.

Attempting to use the legacy include path `esp_adc_cal.h` will trigger the build warning below by default. However, the warning can be suppressed by enabling the *CONFIG_ADC_CALI_SUPPRESS_DEPRECATE_WARN* Kconfig option.

```
legacy adc calibration driver is deprecated, please migrate to use esp_adc/adc_
↪cali.h and esp_adc/adc_cali_scheme.h
```

API Changes

- The ADC power management APIs `adc_power_acquire` and `adc_power_release` have made private and moved to `esp_private/adc_share_hw_ctrl.h`.
  - The two APIs were previously made public due to a HW errata workaround.
  - Now, ADC power management is completely handled internally by drivers.
  - Users who still require this API can include `esp_private/adc_share_hw_ctrl.h` to continue using these functions.
- `driver/adc2_wifi_private.h` has been moved to `esp_private/adc_share_hw_ctrl.h`.
- Enums `ADC_UNIT_BOTH`, `ADC_UNIT_ALTER`, and `ADC_UNIT_MAX` in `adc_unit_t` have been removed.
- The following enumerations have been removed as some of their enumeration values are not supported on all chips. This would lead to the driver triggering a runtime error if an unsupported value is used.
  - Enum `ADC_CHANNEL_MAX`
  - Enum `ADC_ATTEN_MAX`
  - Enum `ADC_CONV_UNIT_MAX`
- API `hall_sensor_read` on ESP32 has been removed. Hall sensor is no longer supported on ESP32.
- API `adc_set_i2s_data_source` and `adc_i2s_mode_init` have been deprecated. Related enum `adc_i2s_source_t` has been deprecated. Please migrate to use `esp_adc/adc_continuous.h`.
- API `adc_digi_filter_reset`, `adc_digi_filter_set_config`, `adc_digi_filter_get_config` and `adc_digi_filter_enable` have been removed. These APIs behaviours are not guaranteed. Enum `adc_digi_filter_idx_t`, `adc_digi_filter_mode_t` and structure `adc_digi_iir_filter_t` have been removed as well.
- API `esp_adc_cal_characterize` has been deprecated, please migrate to `adc_cali_create_scheme_curve_fitting` or `adc_cali_create_scheme_line_fitting` instead.
- API `esp_adc_cal_raw_to_voltage` has been deprecated, please migrate to `adc_cali_raw_to_voltage` instead.
- API `esp_adc_cal_get_voltage` has been deprecated, please migrate to `adc_oneshot_get_calibrated_result` instead.

GPIO

- The previous Kconfig option *RTCIO_SUPPORT_RTC_GPIO_DESC* has been removed, thus the `rtc_gpio_desc` array is unavailable. Please use `rtc_io_desc` array instead.
- The user callback of a GPIO interrupt should no longer read the GPIO interrupt status register to get the GPIO's pin number of triggering the interrupt. You should use the callback argument to determine the GPIO's pin number instead.
  - Previously, when a GPIO interrupt occurs, the GPIO's interrupt status register is cleared after calling the user callbacks. Thus, it was possible for users to read the GPIO's interrupt status register inside the callback to determine which GPIO was used to trigger the interrupt.
  - However, clearing the interrupt status register after calling the user callbacks can potentially cause edge-triggered interrupts to be lost. For example, if an edge-triggered interrupt (re)is triggered while the user callbacks are being called, that interrupt will be cleared without its registered user callback being handled.
  - Now, the GPIO's interrupt status register is cleared **before** invoking the user callbacks. Thus, users can no longer read the GPIO interrupt status register to determine which pin has triggered the interrupt. Instead, users should use the callback argument to pass the pin number.

Sigma-Delta Modulator The Sigma-Delta Modulator driver has been redesigned into *SDM*.

- The new driver implements a factory pattern, where the SDM channels are managed in a pool internally, thus users don't have to fix a SDM channel to a GPIO manually.
- All SDM channels can be allocated dynamically.

Although it's recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/sigmadelta.h`. However, by default, including `driver/sigmadelta.h` will trigger the build warning below. The warning can be suppressed by Kconfig option *CONFIG_SDM_SUPPRESS_DEPRECATE_WARN*.

```
The legacy sigma-delta driver is deprecated, please use driver/sdm.h
```

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- SDM channel representation has changed from `sigmadelta_channel_t` to *sdm_channel_handle_t*, which is an opaque pointer.
- SDM channel configurations are stored in *sdm_config_t* now, instead the previous `sigmadelta_config_t`.
- In the legacy driver, users don't have to set the clock source for SDM channel. But in the new driver, users need to set a proper one in the *sdm_config_t::clk_src*. The available clock sources are listed in the *soc_periph_sdm_clk_src_t*.
- In the legacy driver, users need to set a `prescale` for the channel, which reflects the frequency in which the modulator outputs a pulse. In the new driver, users should use *sdm_config_t::sample_rate_hz*.

Breaking Changes in Usage

- Channel configuration was done by channel allocation, in *sdm_new_channel()*. In the new driver, only the `duty` can be changed at runtime, by *sdm_channel_set_duty()*. Other parameters like `gpio number` and `prescale` are only allowed to set during channel allocation.
- Before further channel operations, users should **enable** the channel in advance, by calling *sdm_channel_enable()*. This function will help to manage some system level services, like **Power Management**.

Timer Group Driver Timer Group driver has been redesigned into *GPTimer*, which aims to unify and simplify the usage of general purpose timer.

Although it's recommended to use the the new driver APIs, the legacy driver is still available in the previous include path `driver/timer.h`. However, by default, including `driver/timer.h` will trigger the build warning below. The warning can be suppressed by the Kconfig option *CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN*.

```
legacy timer group driver is deprecated, please migrate to driver/gptimer.h
```

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- `timer_group_t` and `timer_idx_t` which used to identify the hardware timer are removed from user's code. In the new driver, a timer is represented by *gptimer_handle_t*.
- Definition of timer clock source is moved to *gptimer_clock_source_t*, the previous `timer_src_clk_t` is not used.
- Definition of timer count direction is moved to *gptimer_count_direction_t*, the previous `timer_count_dir_t` is not used.
- Only level interrupt is supported, `timer_intr_t` and `timer_intr_mode_t` are not used.
- Auto-reload is enabled by set the *gptimer_alarm_config_t::auto_reload_on_alarm* flag. `timer_autoreload_t` is not used.

Breaking Changes in Usage

- Timer initialization is done by creating a timer instance from *gptimer_new_timer()*. Basic configurations like clock source, resolution and direction should be set in *gptimer_config_t*. Note that, specific configurations of alarm events are not needed during the installation stage of the driver.

- Alarm event is configured by *gptimer_set_alarm_action()*, with parameters set in the *gptimer_alarm_config_t*.
- Setting and getting count value are done by *gptimer_get_raw_count()* and *gptimer_set_raw_count()*. The driver doesn't help convert the raw value into UTC time-stamp. Instead, the conversion should be done from user's side as the timer resolution is also known to the user.
- The driver will install the interrupt service as well if *gptimer_event_callbacks_t::on_alarm* is set to a valid callback function. In the callback, users do not have to deal with the low level registers (like "clear interrupt status", "re-enable alarm event" and so on). So functions like `timer_group_get_intr_status_in_isr` and `timer_group_get_auto_reload_in_isr` are not used anymore.
- To update the alarm configurations when alarm event happens, one can call *gptimer_set_alarm_action()* in the interrupt callback, then the alarm will be re-enabled again.
- Alarm will always be re-enabled by the driver if *gptimer_alarm_config_t::auto_reload_on_alarm* is set to true.

UART

Removed/Deprecated items	Replacement	Remarks
`uart_isr_register()`	None	UART interrupt handling is implemented by driver itself.
`uart_isr_free()`	None	UART interrupt handling is implemented by driver itself.
`use_ref_tick` in *uart_config_t*	*uart_config_t::source_clk*	Select the clock source.
`uart_enable_pattern_det_intr()`	*uart_enable_pattern_det_baud_intr()*	Enable pattern detection interrupt.

I2C

Removed/Deprecated items	Replacement	Remarks
`i2c_isr_register()`	None	I2C interrupt handling is implemented by driver itself.
`i2c_isr_register()`	None	I2C interrupt handling is implemented by driver itself.
`i2c_opmode_t`	None	It's not used anywhere in esp-idf.

SPI

Removed/Deprecated items	Replacement	Remarks
`spi_cal_clock()`	*spi_get_actual_clock()*	Get SPI real working frequency.

- The internal header file `spi_common_internal.h` has been moved to `esp_private/spi_common_internal.h`.

SDMMC

Removed/Deprecated items	Replacement	Remarks
`sdmmc_host_pullup_en()`	set `SDMMC_SLOT_FLAG_INTERNAL_PULLUP` flag in *sdmmc_slot_config_t::flags*	Enable internal pull up.

LEDC

Removed/Deprecated items	Replacement	Remarks
`bit_num` in *ledc_timer_config_t*	*ledc_timer_config_t::duty_resolution*	Set resolution of the duty cycle.

Pulse Counter Driver Pulse counter driver has been redesigned (see *PCNT*), which aims to unify and simplify the usage of PCNT peripheral.

Although it's recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/pcnt.h`. However, including `driver/pcnt.h` will trigger the build warning below by default. The warning can be suppressed by the Kconfig option *CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN*.

```
legacy pcnt driver is deprecated, please migrate to use driver/pulse_cnt.h
```

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- `pcnt_port_t`, `pcnt_unit_t` and `pcnt_channel_t` which used to identify the hardware unit and channel are removed from user's code. In the new driver, PCNT unit is represented by *`pcnt_unit_handle_t`*, likewise, PCNT channel is represented by *`pcnt_channel_handle_t`*. Both of them are opaque pointers.
- `pcnt_evt_type_t` is not used any more, they have been replaced by a universal **Watch Point Event**. In the event callback *`pcnt_watch_cb_t`*, it's still possible to distinguish different watch points from *`pcnt_watch_event_data_t`*.
- `pcnt_count_mode_t` is replaced by *`pcnt_channel_edge_action_t`*, and `pcnt_ctrl_mode_t` is replaced by *`pcnt_channel_level_action_t`*.

Breaking Changes in Usage

- Previously, the PCNT unit configuration and channel configuration were combined into a single function: `pcnt_unit_config`. They are now split into the two factory APIs: *`pcnt_new_unit()`* and *`pcnt_new_channel()`* respectively.
  - Only the count range is necessary for initializing a PCNT unit. GPIO number assignment has been moved to *`pcnt_new_channel()`*.
  - High/Low control mode and positive/negative edge count mode are set by stand-alone functions: *`pcnt_channel_set_edge_action()`* and *`pcnt_channel_set_level_action()`*.
- `pcnt_get_counter_value` is replaced by *`pcnt_unit_get_count()`*.
- `pcnt_counter_pause` is replaced by *`pcnt_unit_stop()`*.
- `pcnt_counter_resume` is replaced by *`pcnt_unit_start()`*.
- `pcnt_counter_clear` is replaced by *`pcnt_unit_clear_count()`*.
- `pcnt_intr_enable` and `pcnt_intr_disable` are removed. In the new driver, the interrupt is enabled by registering event callbacks *`pcnt_unit_register_event_callbacks()`*.
- `pcnt_event_enable` and `pcnt_event_disable` are removed. In the new driver, the PCNT events are enabled/disabled by adding/removing watch points *`pcnt_unit_add_watch_point()`*, *`pcnt_unit_remove_watch_point()`*.
- `pcnt_set_event_value` is removed. In the new driver, event value is also set when adding watch point by *`pcnt_unit_add_watch_point()`*.
- `pcnt_get_event_value` and `pcnt_get_event_status` are removed. In the new driver, these information are provided by event callback *`pcnt_watch_cb_t`* in the *`pcnt_watch_event_data_t`*.
- `pcnt_isr_register` and `pcnt_isr_unregister` are removed. Register of the ISR handler from user's code is no longer permitted. Users should register event callbacks instead by calling *`pcnt_unit_register_event_callbacks()`*.
- `pcnt_set_pin` is removed and the new driver no longer allows the switching of the GPIO at runtime. If users want to change to other GPIOs, please delete the existing PCNT channel by *`pcnt_del_channel()`* and reinstall with the new GPIO number by *`pcnt_new_channel()`*.
- `pcnt_filter_enable`, `pcnt_filter_disable` and `pcnt_set_filter_value` are replaced by *`pcnt_unit_set_glitch_filter()`*. Meanwhile, `pcnt_get_filter_value` has been removed.
- `pcnt_set_mode` is replaced by *`pcnt_channel_set_edge_action()`* and *`pcnt_channel_set_level_action()`*.
- `pcnt_isr_service_install`, `pcnt_isr_service_uninstall`, `pcnt_isr_handler_add` and `pcnt_isr_handler_remove` are replaced by

pcnt_unit_register_event_callbacks(). The default ISR handler is lazy installed in the new driver.

RMT Driver RMT driver has been redesigned (see *RMT transceiver*), which aims to unify and extend the usage of RMT peripheral.

Although it's recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/rmt.h`. However, including `driver/rmt.h` will trigger the build warning below by default. The warning can be suppressed by the Kconfig option *CONFIG_RMT_SUPPRESS_DEPRECATE_WARN*.

```
The legacy RMT driver is deprecated, please use driver/rmt_tx.h and/or driver/rmt_
↪rx.h
```

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- `rmt_channel_t` which used to identify the hardware channel are removed from user space. In the new driver, RMT channel is represented by *rmt_channel_handle_t*. The channel is dynamically allocated by the driver, instead of designated by user.
- `rmt_item32_t` is replaced by *rmt_symbol_word_t*, which avoids a nested union inside a struct.
- `rmt_mem_t` is removed, as we don't allow users to access RMT memory block (a.k.an RMTMEM) directly. Direct access to RMTMEM doesn't make sense but make mistakes, especially when the RMT channel also connected with a DMA channel.
- `rmt_mem_owner_t` is removed, as the ownership is controlled by driver, not by user anymore.
- `rmt_source_clk_t` is replaced by *rmt_clock_source_t*, and note they're not binary compatible.
- `rmt_data_mode_t` is removed, the RMT memory access mode is configured to always use Non-FIFO and DMA mode.
- `rmt_mode_t` is removed, as the driver has stand alone install functions for TX and RX channels.
- `rmt_idle_level_t` is removed, setting IDLE level for TX channel is available in *rmt_transmit_config_t::eot_level*.
- `rmt_carrier_level_t` is removed, setting carrier polarity is available in *rmt_carrier_config_t::polarity_active_low*.
- `rmt_channel_status_t` and `rmt_channel_status_result_t` are removed, they're not used anywhere.
- Transmitting by RMT channel doesn't expect user to prepare the RMT symbols, instead, user needs to provide an RMT Encoder to tell the driver how to convert user data into RMT symbols.

Breaking Changes in Usage

- Channel installation has been separated for TX and RX channels into *rmt_new_tx_channel()* and *rmt_new_rx_channel()*.
- `rmt_set_clk_div` and `rmt_get_clk_div` are removed. Channel clock configuration can only be done during channel installation.
- `rmt_set_rx_idle_thresh` and `rmt_get_rx_idle_thresh` are removed. In the new driver, the RX channel IDLE threshold is redesigned into a new concept *rmt_receive_config_t::signal_range_max_ns*.
- `rmt_set_mem_block_num` and `rmt_get_mem_block_num` are removed. In the new driver, the memory block number is determined by *rmt_tx_channel_config_t::mem_block_symbols* and *rmt_rx_channel_config_t::mem_block_symbols*.
- `rmt_set_tx_carrier` is removed, the new driver uses *rmt_apply_carrier()* to set carrier behavior.
- `rmt_set_mem_pd` and `rmt_get_mem_pd` are removed. The memory power is managed by the driver automatically.
- `rmt_memory_rw_rst`, `rmt_tx_memory_reset` and `rmt_rx_memory_reset` are removed. Memory reset is managed by the driver automatically.
- `rmt_tx_start` and `rmt_rx_start` are merged into a single function *rmt_enable()*, for both TX and RX channels.

- rmt_tx_stop and rmt_rx_stop are merged into a single function *rmt_disable()*, for both TX and RX channels.
- rmt_set_memory_owner and rmt_get_memory_owner are removed. RMT memory owner guard is added automatically by the driver.
- rmt_set_tx_loop_mode and rmt_get_tx_loop_mode are removed. In the new driver, the loop mode is configured in *rmt_transmit_config_t::loop_count*.
- rmt_set_source_clk and rmt_get_source_clk are removed. Configuring clock source is only possible during channel installation by *rmt_tx_channel_config_t::clk_src* and *rmt_rx_channel_config_t::clk_src*.
- rmt_set_rx_filter is removed. In the new driver, the filter threshold is redesigned into a new concept *rmt_receive_config_t::signal_range_min_ns*.
- rmt_set_idle_level and rmt_get_idle_level are removed. Setting IDLE level for TX channel is available in *rmt_transmit_config_t::eot_level*.
- rmt_set_rx_intr_en, rmt_set_err_intr_en, rmt_set_tx_intr_en, rmt_set_tx_thr_intr_en and rmt_set_rx_thr_intr_en are removed. The new driver doesn't allow user to turn on/off interrupt from user space. Instead, it provides callback functions.
- rmt_set_gpio and rmt_set_pin are removed. The new driver doesn't support to switch GPIO dynamically at runtime.
- rmt_config is removed. In the new driver, basic configuration is done during the channel installation stage.
- rmt_isr_register and rmt_isr_deregister are removed, the interrupt is allocated by the driver itself.
- rmt_driver_install is replaced by *rmt_new_tx_channel()* and *rmt_new_rx_channel()*.
- rmt_driver_uninstall is replaced by *rmt_del_channel()*.
- rmt_fill_tx_items, rmt_write_items and rmt_write_sample are removed. In the new driver, user needs to provide an encoder to "translate" the user data into RMT symbols.
- rmt_get_counter_clock is removed, as the channel clock resolution is configured by user from *rmt_tx_channel_config_t::resolution_hz*.
- rmt_wait_tx_done is replaced by *rmt_tx_wait_all_done()*.
- rmt_translator_init, rmt_translator_set_context and rmt_translator_get_context are removed. In the new driver, the translator has been replaced by the RMT encoder.
- rmt_get_ringbuf_handle is removed. The new driver doesn't use Ringbuffer to save RMT symbols. Instead, the incoming data are saved to the user provided buffer directly. The user buffer can even be mounted to DMA link internally.
- rmt_register_tx_end_callback is replaced by *rmt_tx_register_event_callbacks()*, where user can register *rmt_tx_event_callbacks_t::on_trans_done* event callback.
- rmt_set_intr_enable_mask and rmt_clr_intr_enable_mask are removed, as the interrupt is handled by the driver, user doesn't need to take care of it.
- rmt_add_channel_to_group and rmt_remove_channel_from_group are replaced by RMT sync manager. Please refer to *rmt_new_sync_manager()*.
- rmt_set_tx_loop_count is removed. The loop count in the new driver is configured in *rmt_transmit_config_t::loop_count*.
- rmt_enable_tx_loop_autostop is removed. In the new driver, TX loop auto stop is always enabled if available, it's not configurable anymore.

LCD

- The LCD panel initialization flow is slightly changed. Now the *esp_lcd_panel_init()* won't turn on the display automatically. User needs to call *esp_lcd_panel_disp_on_off()* to manually turn on the display. Note, this is different from turning on backlight. With this breaking change, user can flash a predefined pattern to the screen before turning on the screen. This can help avoid random noise on the screen after a power on reset.
- *esp_lcd_panel_disp_off()* is deprecated, please use *esp_lcd_panel_disp_on_off()* instead.
- dc_as_cmd_phase is removed. The SPI LCD driver currently doesn't support a 9-bit SPI LCD. Please always use a dedicated GPIO to control the LCD D/C line.
- The way to register RGB panel event callbacks has been moved from the esp_lcd_rgb_panel_config_t into a separate API

`esp_lcd_rgb_panel_register_event_callbacks()`. However, the event callback signature is not changed.
- Previous `relax_on_idle` flag in `esp_lcd_rgb_panel_config_t` has been renamed into `esp_lcd_rgb_panel_config_t::refresh_on_demand`, which expresses the same meaning but with a clear name.
- If the RGB LCD is created with the `refresh_on_demand` flag enabled, the driver will not start a refresh in the *`esp_lcd_panel_draw_bitmap()`*. Now users have to call `esp_lcd_rgb_panel_refresh()` to refresh the screen by themselves.
- `esp_lcd_color_space_t` is deprecated, please use *`lcd_color_space_t`* to describe the color space, and use *`lcd_rgb_element_order_t`* to describe the data order of RGB color.

MCPWM MCPWM driver was redesigned (see *MCPWM*), meanwhile, the legacy driver is deprecated.

The new driver's aim is to make each MCPWM submodule independent to each other, and give the freedom of resource connection back to users.

Although it's recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/mcpwm.h`. However, using legacy driver will rigger the build warning below by default. This warning can be suppressed by the Kconfig option *CONFIG_MCPWM_SUPPRESS_DEPRECATE_WARN*.

```
legacy MCPWM driver is deprecated, please migrate to the new driver (include␣
↪driver/mcpwm_prelude.h)
```

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts The new MCPWM driver is object-oriented, where most of the MCPWM submodule has a driver object associated with it. The driver object is created by factory function like *`mcpwm_new_timer()`*. IO control function always needs an object handle, in the first place.

The legacy driver has an inappropriate assumption, that is the MCPWM operator should be connected to different MCPWM timer. In fact, the hardware doesn't have such limitation. In the new driver, a MCPWM timer can be connected to multiple operators, so that the operators can achieve the best synchronization performance.

The legacy driver presets the way to generate a PWM waveform into a so called `mcpwm_duty_type_t`. However, the duty cycle modes listed there are far from sufficient. Likewise, legacy driver has several preset `mcpwm_deadtime_type_t`, which also doesn't cover all the use cases. What's more, user usually gets confused by the name of the duty cycle mode and dead-time mode. In the new driver, there're no such limitation, but user has to construct the generator behavior from scratch.

In the legacy driver, the ways to synchronize the MCPWM timer by GPIO, software and other timer module are not unified. It increased learning costs for users. In the new driver, the synchronization APIs are unified.

The legacy driver has mixed the concepts of "Fault detector" and "Fault handler". Which make the APIs very confusing to users. In the new driver, the fault object just represents a failure source, and we introduced a new concept –**brake** to express the concept of "Fault handler". What's more, the new driver supports software fault.

The legacy drive only provides callback functions for the capture submodule. The new driver provides more useful callbacks for various MCPWM submodules, like timer stop, compare match, fault enter, brake, etc.

- `mcpwm_io_signals_t` and `mcpwm_pin_config_t` are not used. GPIO configuration has been moved into submodule's configuration structure.
- `mcpwm_timer_t`, `mcpwm_generator_t` are not used. Timer and generator are represented by *`mcpwm_timer_handle_t`* and *`mcpwm_gen_handle_t`*.
- `mcpwm_fault_signal_t` and `mcpwm_sync_signal_t` are not used. Fault and sync source are represented by *`mcpwm_fault_handle_t`* and *`mcpwm_sync_handle_t`*.
- `mcpwm_capture_signal_t` is not used. A capture channel is represented by *`mcpwm_cap_channel_handle_t`*.

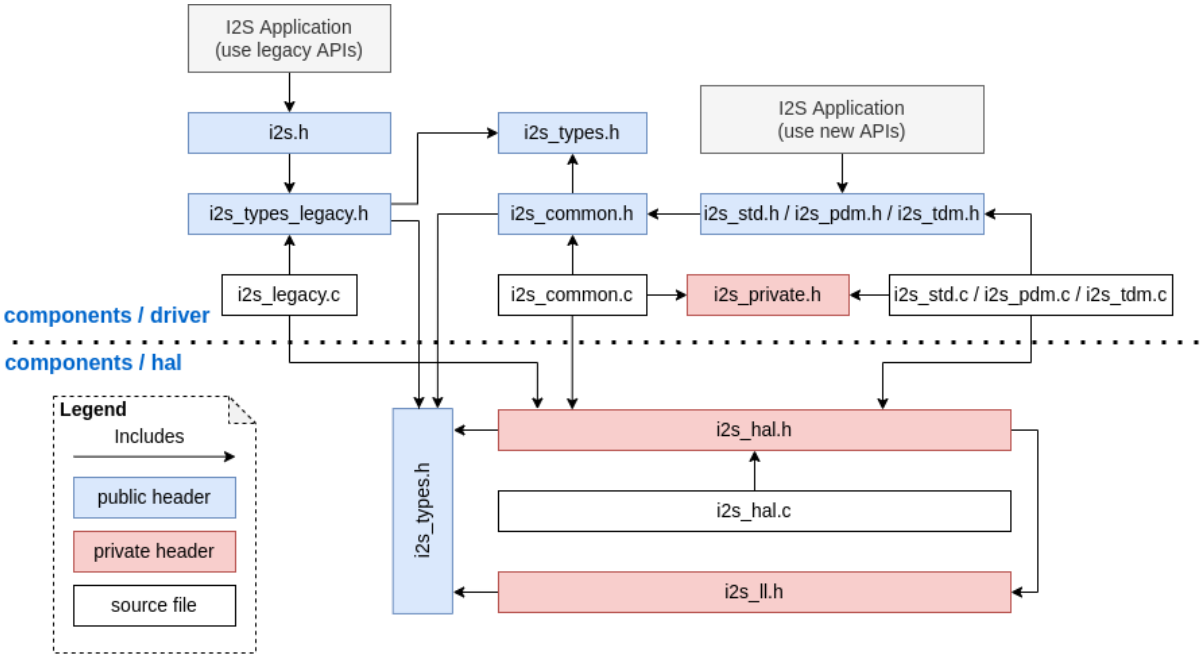
Breaking Changes in Usage

- mcpwm_gpio_init and mcpwm_set_pin: GPIO configurations are moved to submodule's own configuration. e.g. set the PWM GPIO in *mcpwm_generator_config_t::gen_gpio_num*.
- mcpwm_init: To get an expected PWM waveform, users need to allocated at least one MCPWM timer and MCPWM operator, then connect them by calling *mcpwm_operator_connect_timer()*. After that, users should set the generator's actions on various events by calling e.g. *mcpwm_generator_set_actions_on_timer_event()*, *mcpwm_generator_set_actions_on_compare_event()*.
- mcpwm_group_set_resolution: in the new driver, the group resolution is fixed to the maximum, usually it's 80MHz.
- mcpwm_timer_set_resolution: MCPWM Timer resolution is set in *mcpwm_timer_config_t::resolution_hz*.
- mcpwm_set_frequency: PWM frequency is determined by *mcpwm_timer_config_t::resolution_hz*, *mcpwm_timer_config_t::count_mode* and *mcpwm_timer_config_t::period_ticks*.
- mcpwm_set_duty: To set the PWM duty cycle, users should call *mcpwm_comparator_set_compare_value()* to change comparator's threshold.
- mcpwm_set_duty_type: There won't be any preset duty cycle types, the duty cycle type is configured by setting different generator actions. e.g. *mcpwm_generator_set_actions_on_timer_event()*.
- mcpwm_set_signal_high and mcpwm_set_signal_low are replaced by *mcpwm_generator_set_force_level()*. In the new driver, it's implemented by setting force action for the generator, instead of changing the duty cycle to 0% or 100% at the background.
- mcpwm_start and mcpwm_stop are replaced by *mcpwm_timer_start_stop()*. You have more modes to start and stop the MCPWM timer, see *mcpwm_timer_start_stop_cmd_t*.
- mcpwm_carrier_init is replaced by *mcpwm_operator_apply_carrier()*.
- mcpwm_carrier_enable and mcpwm_carrier_disable: Enabling and disabling carrier submodule is done automatically by checking whether the carrier configuration structure *mcpwm_carrier_config_t* is NULL.
- mcpwm_carrier_set_period is replaced by *mcpwm_carrier_config_t::frequency_hz*.
- mcpwm_carrier_set_duty_cycle is replaced by *mcpwm_carrier_config_t::duty_cycle*.
- mcpwm_carrier_oneshot_mode_enable is replaced by *mcpwm_carrier_config_t::first_pulse_durat*
- mcpwm_carrier_oneshot_mode_disable is removed. Disabling the first pulse (a.k.a the one-shot pulse) in the carrier is never supported by the hardware.
- mcpwm_carrier_output_invert is replaced by *mcpwm_carrier_config_t::invert_before_modulate* and *mcpwm_carrier_config_t::invert_after_modulate*.
- mcpwm_deadtime_enable and mcpwm_deadtime_disable are replaced by *mcpwm_generator_set_dead_time()*.
- mcpwm_fault_init is replaced by *mcpwm_new_gpio_fault()*.
- mcpwm_fault_set_oneshot_mode, mcpwm_fault_set_cyc_mode are replaced by *mcpwm_operator_set_brake_on_fault()* and *mcpwm_generator_set_actions_on_brake_event()*.
- mcpwm_capture_enable is removed. It's duplicated to mcpwm_capture_enable_channel().
- mcpwm_capture_disable is removed. It's duplicated to mcpwm_capture_capture_disable_channel().
- mcpwm_capture_enable_channel and mcpwm_capture_disable_channel are replaced by *mcpwm_capture_channel_enable()* and *mcpwm_capture_channel_disable()*.
- mcpwm_capture_signal_get_value and mcpwm_capture_signal_get_edge: Capture timer count value and capture edge are provided in the capture event callback, via *mcpwm_capture_event_data_t*. Capture data are only valuable when capture event happens. Providing single API to fetch capture data is meaningless.
- mcpwm_sync_enable is removed. It's duplicated to mcpwm_sync_configure().
- mcpwm_sync_configure is replaced by *mcpwm_timer_set_phase_on_sync()*.
- mcpwm_sync_disable is equivalent to setting *mcpwm_timer_sync_phase_config_t::sync_src* to NULL.
- mcpwm_set_timer_sync_output is replaced by *mcpwm_new_timer_sync_src()*.
- mcpwm_timer_trigger_soft_sync is replaced by *mcpwm_soft_sync_activate()*.
- mcpwm_sync_invert_gpio_synchro is equivalent to setting *mcpwm_gpio_sync_src_config_t::active_ne*
- mcpwm_isr_register is removed. You can register various event callbacks instead. For example, to register capture event callback, users can use *mcpwm_capture_channel_register_event_callbacks()*.

I2S driver The I2S driver has been redesigned (see *I2S Driver*), which aims to rectify the shortcomings of the driver that were exposed when supporting all the new features of ESP32-C3 & ESP32-S3. The new driver's APIs are available by including corresponding I2S mode's header files driver/include/driver/i2s_std.h, driver/include/driver/i2s_pdm.h, or driver/include/driver/i2s_tdm.h.

Meanwhile, the old driver's APIs in driver/deprecated/driver/i2s.h are still supported for backward compatibility. But there will be warnings if users keep using the old APIs in their projects, these warnings can be suppressed by the Kconfig option *CONFIG_I2S_SUPPRESS_DEPRECATE_WARN*.

Here is the general overview of the current I2S files:



Breaking changes in Concepts

Independent TX/RX channels The minimum control unit in new I2S driver are now individual TX/RX channels instead of an entire I2S controller (that consistes of multiple channels).

- The TX and RX channels of the same I2S controller can be controlled separately, meaning that they are configured such that they can be started or stopped separately.
- The c:type:*i2s_chan_handle_t* handle type is used to uniquely identify I2S channels. All the APIs will require the channel handle and users need to maintain the channel handles by themselves.
- On the ESP32-C3 and ESP32-S3, TX and RX channels in the same controller can be configured to different clocks or modes.
- However, on the ESP32 and ESP32-S2, the TX and RX channels of the same controller still share some hardware resources. Thus, configurations may cause one channel to affect another channel in the same controller.
- The channels can be registered to an available I2S controller automatically by setting `i2s_port_t::I2S_NUM_AUTO` as I2S port ID which will cause the driver to search for the available TX/RX channels. However, the driver also supports registering channels to a specific port.
- In order to distinguish between TX/RX channels and sound channels, the term 'channel' in the context of the I2S driver will only refer to TX/RX channels. Meanwhile, sound channels will be referred to as "slots".

I2S Mode Categorization I2S communication modes are categorized into the following three modes. Note that:

- **Standard mode**: Standard mode always has two slots, it can support Philips, MSB, and PCM (short frame sync) formats. Please refer to driver/include/driver/i2s_std.h for more details.
- **PDM mode**: PDM mode only supports two slots with 16-bit data width, but the configurations of PDM TX and PDM RX are slightly different. For PDM TX, the sample rate

can be set by `i2s_pdm_tx_clk_config_t::sample_rate`, and its clock frequency depends on the up-sampling configuration. For PDM RX, the sample rate can be set by `i2s_pdm_rx_clk_config_t::sample_rate`, and its clock frequency depends on the down-sampling configuration. Please refer to driver/include/driver/i2s_pdm.h for details.

- **TDM mode**: TDM mode can support up to 16 slots. It can work in Philips, MSB, PCM (short frame sync), and PCM (long frame sync) formats. Please refer to driver/include/driver/i2s_tdm.h for details.

When allocating a new channel in a specific mode, users should initialize that channel by its corresponding function. It is strongly recommended to use the helper macros to generate the default configurations in case the default values are changed in the future.

Independent Slot and Clock Configuration The slot configurations and clock configurations can be configured separately.

- Call *i2s_channel_init_std_mode()*, *i2s_channel_init_pdm_rx_mode()*, *i2s_channel_init_pdm_tx_mode()*, or `i2s_channel_init_tdm_mode()` to initialize the slot/clock/gpio_pin configurations.
- Calling *i2s_channel_reconfig_std_slot()*, *i2s_channel_reconfig_pdm_rx_slot()*, *i2s_channel_reconfig_pdm_tx_slot()*, or `i2s_channel_reconfig_tdm_slot()` can change the slot configurations after initialization.
- Calling *i2s_channel_reconfig_std_clock()*, *i2s_channel_reconfig_pdm_rx_clock()*, *i2s_channel_reconfig_pdm_tx_clock()*, or `i2s_channel_reconfig_tdm_clock()` can change the clock configurations after initialization.
- Calling *i2s_channel_reconfig_std_gpio()*, *i2s_channel_reconfig_pdm_rx_gpio()*, *i2s_channel_reconfig_pdm_tx_gpio()*, or `i2s_channel_reconfig_tdm_gpio()` can change the GPIO configurations after initialization.

Misc

- States and state-machine are adopted in the new I2S driver to avoid APIs called in wrong state.
- ADC and DAC modes are removed. They will only be supported in their own drivers and the legacy I2S driver.

Breaking Changes in Usage To use the new I2S driver, please follow these steps:

1. Call *i2s_new_channel()* to acquire channel handles. We should specify the work role and I2S port in this step. Besides, the TX or RX channel handle will be generated by the driver. Inputting both two TX and RX channel handles is not necessary but at least one handle is needed. In the case of inputting both two handles, the driver will work at the duplex mode. Both TX and RX channels will be avaliable on a same port, and they will share the MCLK, BCLK and WS signal. But if only one of the TX or RX channel handle is inputted, this channel will only work in the simplex mode.
2. Call `i2s_channel_init_std_mode()`, `i2s_channel_init_pdm_rx_mode()`, `i2s_channel_init_pdm_tx_mode()` or `i2s_channel_init_tdm_mode()` to initialize the channel to the specified mode. Corresponding slot, clock and GPIO configurations are needed in this step.
3. (Optional) Call *i2s_channel_register_event_callback()* to register the ISR event callback functions. I2S events now can be received by the callback function synchronously, instead of from the event queue asynchronously.
4. Call *i2s_channel_enable()* to start the hardware of I2S channel. In the new driver, I2S won't start automatically after installed, and users are supposed to know clearly whether the channel has started or not.
5. Read or write data by *i2s_channel_read()* or *i2s_channel_write()*. Certainly, only the RX channel handle is suppoesd to be inputted in *i2s_channel_read()* and the TX channel handle in *i2s_channel_write()*.
6. (Optional) The slot, clock and GPIO configurations can be changed by corresponding 'reconfig' functions, but *i2s_channel_disable()* must be called before updating the configurations.
7. Call *i2s_channel_disable()* to stop the hardware of I2S channel.
8. Call *i2s_del_channel()* to delete and release the resources of the channel if it is not needed any more, but the channel must be disabled before deleting it.

TWAI Driver The deprecated `CAN` peripheral driver is removed. Please use `TWAI` driver instead (i.e. include `driver/twai.h` in your application).

Register Access Macros Previously, all register access macros could be used as expressions, so the following was allowed:

```
uint32_t val = REG_SET_BITS(reg, mask);
```

In ESP-IDF v5.0, register access macros which write or read-modify-write the register can no longer be used as expressions, and can only be used as statements. This applies to the following macros: `REG_WRITE`, `REG_SET_BIT`, `REG_CLR_BIT`, `REG_SET_BITS`, `REG_SET_FIELD`, `WRITE_PERI_REG`, `CLEAR_PERI_REG_MASK`, `SET_PERI_REG_MASK`, `SET_PERI_REG_BITS`.

To store the value which would have been written into the register, split the operation as follows:

```
uint32_t new_val = REG_READ(reg) | mask;
REG_WRITE(reg, new_val);
```

To get the value of the register after modification (which may be different from the value written), add an explicit read:

```
REG_SET_BITS(reg, mask);
uint32_t new_val = REG_READ(reg);
```

Protocols

Mbed TLS For ESP-IDF v5.0, Mbed TLS has been updated from v2.x to v3.1.0.

For more details about Mbed TLS' s migration from version 2.x to version 3.0 or greater, please refer to the official guide.

Breaking Changes (Summary)

Most structure fields are now private

- Direct access to fields of structures (`struct` types) declared in public headers is no longer supported.
- Appropriate accessor functions (getter/setter) must be used for the same. A temporary workaround would be to use `MBEDTLS_PRIVATE` macro (**not recommended**).
- For more details, refer to the official guide.

SSL

- Removed support for TLS 1.0, 1.1, and DTLS 1.0
- Removed support for SSL 3.0

Deprecated Functions Were Removed from Cryptography Modules

- The functions `mbedtls_*_ret()` (related to MD, SHA, RIPEMD, RNG, HMAC modules) was renamed to replace the corresponding functions without `_ret` appended and updated return value.
- For more details, refer to the official guide.

Deprecated Config Options Following are some of the important config options deprecated by this update. The configs related to and/or dependent on these have also been deprecated.

- `MBEDTLS_SSL_PROTO_SSL3` : Support for SSL 3.0
- `MBEDTLS_SSL_PROTO_TLS1` : Support for TLS 1.0
- `MBEDTLS_SSL_PROTO_TLS1_1`: Support for TLS 1.1
- `MBEDTLS_SSL_PROTO_DTLS` : Support for DTLS 1.1 (Only DTLS 1.2 is supported now)
- `MBEDTLS_DES_C` : Support for 3DES ciphersuites
- `MBEDTLS_RC4_MODE` : Support for RC4-based ciphersuites

Note: This list includes only major options configurable through `idf.py menuconfig`. For more details on deprecated options, refer to the official guide.

Miscellaneous

Disabled Diffie-Hellman Key Exchange Modes The Diffie-Hellman Key Exchange modes have now been disabled by default due to security risks (see warning text here). Related configs are given below:

- `MBEDTLS_DHM_C` : Support for the Diffie-Hellman-Merkle module
- `MBEDTLS_KEY_EXCHANGE_DHE_PSK` : Support for Diffie-Hellman PSK (pre-shared-key) TLS authentication modes
- `MBEDTLS_KEY_EXCHANGE_DHE_RSA` : Support for cipher suites with the prefix `TLS-DHE-RSA-WITH-`

Note: During the initial step of the handshake (i.e. `client_hello`), the server selects a cipher from the list that the client publishes. As the DHE_PSK/DHE_RSA ciphers have now been disabled by the above change, the server would fall back to an alternative cipher; if in a rare case, it does not support any other cipher, the handshake would fail. To retrieve the list of ciphers supported by the server, one must attempt to connect with the server with a specific cipher from the client-side. Few utilities can help do this, e.g. `sslscan`.

Remove `certs` Module from X509 Library

- The `mbedtls/certs.h` header is no longer available in mbedtls 3.1. Most applications can safely remove it from the list of includes.

Breaking Change for `esp_crt_bundle_set` API

- The *`esp_crt_bundle_set()`* API now requires one additional argument named `bundle_size`. The return type of the API has also been changed to *`esp_err_t`* from `void`.

Breaking Change for `esp_ds_rsa_sign` API

- The `esp_ds_rsa_sign()` API now requires one less argument. The argument `mode` is no longer required.

HTTPS Server

Breaking Changes (Summary) Names of variables holding different certs in *`httpd_ssl_config_t`* structure have been updated.

- *`httpd_ssl_config::servercert`* variable inherits role of `cacert_pem` variable.
- *`httpd_ssl_config::servercert_len`* variable inherits role of `cacert_len` variable
- *`httpd_ssl_config::cacert_pem`* variable inherits role of `client_verify_cert_pem` variable

- *httpd_ssl_config::cacert_len* variable inherits role of `client_verify_cert_len` variable

The return type of the *httpd_ssl_stop()* API has been changed to *esp_err_t* from `void`.

ESP HTTPS OTA

Breaking Changes (Summary)

- The function *esp_https_ota()* now requires pointer to *esp_https_ota_config_t* as argument instead of pointer to *esp_http_client_config_t*.

ESP-TLS

Breaking Changes (Summary)

`esp_tls_t` Structure is Now Private The *esp_tls_t* has now been made completely private. You cannot access its internal structures directly. Any necessary data that needs to be obtained from the ESP-TLS handle can be done through respective getter/setter functions. If there is a requirement of a specific getter/setter function, please raise an issue on ESP-IDF.

The list of newly added getter/setter function is as as follows:

- *esp_tls_get_ssl_context()* - Obtain the ssl context of the underlying ssl stack from the ESP-TLS handle.

Function Deprecations And Recommended Alternatives Following table summarizes the deprecated functions removed and their alternatives to be used from ESP-IDF v5.0 onwards.

Deprecated Function	Alternative
`esp_tls_conn_new()`	*esp_tls_conn_new_sync()*
`esp_tls_conn_delete()`	*esp_tls_conn_destroy()*

- The function *esp_tls_conn_http_new()* has now been termed as deprecated. Please use the alternative function *esp_tls_conn_http_new_sync()* (or its asynchronous *esp_tls_conn_http_new_async()*). Note that the alternatives need an additional parameter *esp_tls_t*, which has to be initialized using the *esp_tls_init()* function.

HTTP Server

Breaking Changes (Summary)

- `http_server.h` header is no longer available in `esp_http_server`. Please use `esp_http_server.h` instead.

ESP HTTP Client

Breaking Changes (Summary)

- The functions *esp_http_client_read()* and *esp_http_client_fetch_headers()* now return an additional return value `-ESP_ERR_HTTP_EAGAIN` for timeout errors - call timed-out before any data was ready.

TCP Transport

Breaking Changes (Summary)

- The function `esp_transport_read()` now returns `0` for a connection timeout and `< 0` for other errors. Please refer `esp_tcp_transport_err_t` for all possible return values.

MQTT Client

Breaking Changes (Summary)

- *`esp_mqtt_client_config_t`* have all fields grouped in sub structs.

Most common configurations are listed below:

- Broker address now is set in `esp_mqtt_client_config_t::broker::address::uri`
- Security related to broker verification in `esp_mqtt_client_config_t::broker::verification`
- Client username is set in `esp_mqtt_client_config_t::credentials::username`
- *`esp_mqtt_client_config_t`* no longer supports the `user_context` field. Please use *`esp_mqtt_client_register_event()`* instead for registering an event handler; the last argument `event_handler_arg` can be used to pass user context to the handler.

ESP-Modbus

Breaking Changes (Summary) The ESP-IDF component `freemodbus` has been removed from ESP-IDF and is supported as a separate component. Additional information for the `ESP-Modbus` component can be found in the separate repository:

- ESP-Modbus component on GitHub

The `main` component folder of the new application shall include the component manager manifest file `idf_component.yml` as in the example below:

```
dependencies:
  espressif/esp-modbus:
    version: "^1.0"
```

The `esp-modbus` component can be found in component manager registry. Refer to component manager documentation for more information on how to set up the component manager.

For applications targeting v4.x releases of ESP-IDF that need to use new `esp-modbus` component, adding the component manager manifest file `idf_component.yml` will be sufficient to pull in the new component. However, users should also exclude the legacy `freemodbus` component from the build. This can be achieved using the statement below in the project's `CMakeLists.txt`:

```
set(EXCLUDE_COMPONENTS freemodbus)
```

Provisioning

Protocomm The `pop` field in the *`protocomm_set_security()`* API is now deprecated. Please use the `sec_params` field instead of `pop`. This parameter should contain the structure (including the security parameters) as required by the protocol version used.

For example, when using security version 2, the `sec_params` parameter should contain the pointer to the structure of type *`protocomm_security2_params_t`*.

Wi-Fi Provisioning

- The `pop` field in the *`wifi_prov_mgr_start_provisioning()`* API is now deprecated. For backward compatibility, `pop` can be still passed as a string for security1. However for Security2 the `wifi_prov_sec_params` argument needs to be passed instead of `pop`. This parameter should contain the structure (containing the security parameters) as required by the protocol version used. For example, when using security version 2, the `wifi_prov_sec_params` parameter should contain the pointer to the structure of type *`wifi_prov_security2_params_t`*. For security 1 the behaviour and the usage of the API remains same.
- The API *`wifi_prov_mgr_is_provisioned()`* does not return *`ESP_ERR_INVALID_STATE`* error any more. This API now works without any dependency on provisioning manager initialization state.

ESP Local Control The `pop` field in the *`esp_local_ctrl_proto_sec_cfg_t`* API is now deprecated. Please use the `sec_params` field instead of `pop`. This field should contain the structure (containing the security parameters) as required by the protocol version used.

For example, when using security version 2, the `sec_params` field should contain pointer to the structure of type *`esp_local_ctrl_security2_params_t`*.

Removed or Deprecated Components

Components Moved to IDF Component Registry Following components are removed from ESP-IDF and moved to IDF Component Registry:

- libsodium
- cbor
- jsmn
- esp_modem
- nghttp
- mdns
- esp_websocket_client
- asio
- freemodbus
- sh2lib
- expat
- coap
- esp-cryptoauthlib
- qrcode
- tjpgd
- esp_serial_slave_link
- tinyusb

Note: Please note that http parser functionality which was previously part of `nghttp` component is now part of http_parser component.

These components can be installed using `idf.py add-dependency` command.

For example, to install libsodium component with the exact version X.Y, run `idf.py add-dependency libsodium==X.Y`.

To install libsodium component with the latest version compatible to X.Y according to semver rules, run `idf.py add-dependency libsodium~X.Y`.

To find out which versions of each component are available, open https://components.espressif.com, search for the component by its name and check the versions listed on the component page.

Deprecated Components The following components are removed since they were deprecated in IDF v4.x:

- `tcpip_adapter`. Please use the *ESP-NETIF* component instead; you can follow the *TCP/IP Adapter*.

Note: OpenSSL-API component is no longer supported. It is not available in the IDF Component Registry, either. Please use *ESP-TLS* or mbedtls API directly.

The targets components are no longer necessary after refactoring and have been removed:

- `esp32`
- `esp32s2`
- `esp32s3`
- `esp32c2`
- `esp32c3`
- `esp32h2`

Storage

New Component for the Partition APIs Breaking change: all the Partition API code has been moved to a new component esp_partition. For the complete list of affected functions and data-types, see header file esp_partition.h .

These API functions and data-types were previously a part of the spi_flash component, and thus possible dependencies on the `spi_flash` in existing applications may cause the build failure, in case of direct esp_partition_* APIs/data-types use (for instance, `fatal error: esp_partition.h: No such file or directory` at lines with `#include "esp_partition.h"`). If you encounter such an issue, please update your project's CMakeLists.txt file as follows:

Original dependency setup:

```
idf_component_register(...
                       REQUIRES spi_flash)
```

Updated dependency setup:

```
idf_component_register(...
                       REQUIRES spi_flash esp_partition)
```

Note: Please update relevant `REQUIRES` or `PRIV_REQUIRES` section according to your project. The above-presented code snippet is just an example.

If the issue persists, please let us know and we will assist you with your code migration.

SDMMC/SDSPI SD card frequency on SDMMC/SDSPI interface can be now configured through `sdmmc_host_t.max_freq_khz` to a specific value, not only `SDMMC_FREQ_PROBING` (400 kHz), `SDMMC_FREQ_DEFAULT` (20 MHz), or `SDMMC_FREQ_HIGHSPEED` (40 MHz). Previously, in case you have specified a custom frequency other than any of the above-mentioned values, the closest lower-or-equal one was selected anyway.

Now, the underlaying drivers calculate the nearest fitting value, given by available frequency dividers instead of an enumeration item selection. This could cause troubles in communication with your SD card without a change of the existing application code.If you encounter such an issue, please, keep trying different frequencies around your desired value unless you find the one working well. To check the frequency value calculated and actually applied, use `void sdmmc_card_print_info(FILE* stream, const sdmmc_card_t* card)` function.

FatFs FatFs is now updated to v0.14. As a result, the function signature of `f_mkfs()` has changed. The new signature is `FRESULT f_mkfs (const TCHAR* path, const MKFS_PARM* opt, void* work, UINT len);` which uses `MKFS_PARM` struct as a second argument.

Partition Table The partition table generator no longer supports misaligned partitions. When generating a partition table, `ESP-IDF` only accepts partitions with offsets that align to 4 KB. This change only affects generating new partition tables. Reading and writing to already existing partitions remains unchanged.

VFS The `esp_vfs_semihost_register()` function signature is changed as follows:

- The new signature is `esp_err_t esp_vfs_semihost_register(const char* base_path);`
- The `host_path` parameter of the old signature no longer exists. Instead, the OpenOCD command `ESP_SEMIHOST_BASEDIR` should be used to set the full path on the host.

Function Signature Changes The following functions now return `esp_err_t` instead of `void` or `nvs_iterator_t`. Previously, when parameters were invalid or when something goes wrong internally, these functions would `assert()` or return a `nullptr`. With an `esp_err_t` returned, you can get better error reporting.

- *`nvs_entry_find()`*
- *`nvs_entry_next()`*
- *`nvs_entry_info()`*

Because the `esp_err_t` return type changes, the usage patterns of `nvs_entry_find()` and `nvs_entry_next()` become different. Both functions now modify iterators via parameters instead of returning an iterator.

The old programming pattern to iterate over an NVS partition was as follows:

```
nvs_iterator_t it = nvs_entry_find(<nvs_partition_name>, <namespace>, NVS_TYPE_
↪ANY);
while (it != NULL) {
        nvs_entry_info_t info;
        nvs_entry_info(it, &info);
        it = nvs_entry_next(it);
        printf("key '%s', type '%d'", info.key, info.type);
};
```

The new programming pattern to iterate over an NVS partition is now:

```
nvs_iterator_t it = nullptr;
esp_err_t res = nvs_entry_find(<nvs_partition_name>, <namespace>, NVS_TYPE_ANY, &
↪it);
while(res == ESP_OK) {
    nvs_entry_info_t info;
    nvs_entry_info(it, &info); // Can omit error check if parameters are␣
↪guaranteed to be non-NULL
    printf("key '%s', type '%d'", info.key, info.type);
    res = nvs_entry_next(&it);
}
nvs_release_iterator(it);
```

Iterator Validity Note that because the function signature changes, if there is a parameter error, you may get an invalid iterator from `nvs_entry_find()`. Hence, it is important to initialize the iterator to `NULL` before using `nvs_entry_find()`, so that you can avoid complex error checking before calling `nvs_release_iterator()`. A good example is the programming pattern above.

Removed SDSPI Deprecated API Structure `sdspi_slot_config_t` and function `sdspi_host_init_slot()` are removed, and replaced by structure `sdspi_device_config_t` and function `sdspi_host_init_device()` respectively.

ROM SPI Flash In versions before v5.0, ROM SPI flash functions were included via `esp32**/rom/spi_flash.h`. Thus, code written to support different ESP chips might be filled with ROM headers of different targets. Furthermore, not all of the APIs could be used on all ESP chips.

Now, the common APIs are extracted to `esp_rom_spiflash.h`. Although it is not a breaking change, you are strongly recommended to only use the functions from this header (i.e., prefixed with `esp_rom_spiflash` and included by `esp_rom_spiflash.h`) for better cross-compatibility between ESP chips.

To make ROM SPI flash APIs clearer, the following functions are also renamed:

- `esp_rom_spiflash_lock()` to `esp_rom_spiflash_set_bp()`
- `esp_rom_spiflash_unlock()` to `esp_rom_spiflash_clear_bp()`

SPI Flash Driver The `esp_flash_speed_t enum` type is now deprecated. Instead, you may now directly pass the real clock frequency value to the flash configuration structure. The following example demonstrates how to configure a flash frequency of 80MHz:

```
esp_flash_spi_device_config_t dev_cfg = {
    // Other members
    .freq_mhz = 80,
    // Other members
};
```

Legacy SPI Flash Driver To make SPI flash drivers more stable, the legacy SPI flash driver is removed from v5.0. The legacy SPI flash driver refers to default spi_flash driver since v3.0, and the SPI flash driver with configuration option `CONFIG_SPI_FLASH_USE_LEGACY_IMPL` enabled since v4.0. The major breaking change here is that the legacy spi_flash driver is no longer supported from v5.0. Therefore, the legacy driver APIs and the `CONFIG_SPI_FLASH_USE_LEGACY_IMPL` configuration option are both removed. Please use the new spi_flash driver's APIs instead.

Removed items	Replacement
`spi_flash_erase_sector()`	`esp_flash_erase_region()`
`spi_flash_erase_range()`	`esp_flash_erase_region()`
`spi_flash_write()`	`esp_flash_write()`
`spi_flash_read()`	`esp_flash_read()`
`spi_flash_write_encrypted()`	`esp_flash_write_encrypted()`
`spi_flash_read_encrypted()`	`esp_flash_read_encrypted()`

Note: New functions with prefix `esp_flash` accept an additional `esp_flash_t*` parameter. You can simply set it to NULL. This will make the function to run the main flash (`esp_flash_default_chip`).

The `esp_spi_flash.h` header is deprecated as system functions are no longer public. To use flash memory mapping APIs, you may include `spi_flash_mmap.h` instead.

System

Inter-Processor Call IPC (Inter-Processor Call) feature is no longer a stand-alone component and has been integrated into the `esp_system` component.

Thus, any project presenting a `CMakeLists.txt` file with the parameters `PRIV_REQUIRES esp_ipc` or `REQUIRES esp_ipc` should be modified to simply remove these options as the `esp_system` component is included by default.

ESP Clock The ESP Clock API (functions/types/macros prefixed with `esp_clk`) has been made into a private API. Thus, the previous include paths `#include "ESP32/clk.h"` and `#include "esp_clk.h"` have been removed. If users still require usage of the ESP Clock API (though this is not recommended), it can be included via `#include "esp_private/esp_clk.h"`.

Note: Private APIs are not stable and no are longer subject to the ESP-IDF versioning scheme's breaking change rules. Thus, it is not recommended for users to continue calling private APIs in their applications.

Cache Error Interrupt The Cache Error Interrupt API (functions/types/macros prefixed with `esp_cache_err`) has been made into a private API. Thus, the previous include path `#include "ESP32/cache_err_int.h"` has been removed. If users still require usage of the Cache Error Interrupt API (though this is not recommended), it can be included via `#include "esp_private/cache_err_int.h"`.

bootloader_support

- The function `bootloader_common_get_reset_reason()` has been removed. Please use the function `esp_rom_get_reset_reason()` in the ROM component.
- The functions `esp_secure_boot_verify_sbv2_signature_block()` and `esp_secure_boot_verify_rsa_signature_block()` have been removed without replacement. We do not expect users to use these directly. If they are indeed still neccessary, please open a feature request on GitHub explaining why these functions are necessary to you.

Brownout The Brownout API (functions/types/macros prefixed with `esp_brownout`) has been made into a private API. Thus, the previous include path `#include "brownout.h"` has been removed. If users still require usage of the Brownout API (though this is not recommended), it can be included via `#include "esp_private/brownout.h"`.

Trax The Trax API (functions/types/macros prefixed with `trax_`) has been made into a private API. Thus, the previous include path `#include "trax.h"` has been removed. If users still require usage of the Trax API (though this is not recommended), it can be included via `#include "esp_private/trax.h"`.

ROM The previously deprecated ROM-related header files located in `components/esp32/rom/` (old include path: `rom/*.h`) have been moved. Please use the new target-specific path from `components/esp_rom/include/ESP32/` (new include path: `ESP32/rom/*.h`).

esp_hw_support

- The header files `soc/cpu.h` have been deleted and deprecated CPU util functions have been removed. ESP-IDF developers should include `esp_cpu.h` instead for equivalent functions.
- The header files `hal/cpu_ll.h`, `hal/cpu_hal.h`, `hal/soc_ll.h`, `hal/soc_hal.h` and `interrupt_controller_hal.h` CPU API functions have been deprecated. ESP-IDF developers should include `esp_cpu.h` instead for equivalent functions.
- The header file `compare_set.h` have been deleted. ESP-IDF developers should use `esp_cpu_compare_and_set()` function provided in `esp_cpu.h` instead.
- `esp_cpu_get_ccount()`, `esp_cpu_set_ccount()` and `esp_cpu_in_ocd_debug_mode()` were removed from `esp_cpu.h`. ESP-IDF developers should use respectively `esp_cpu_get_cycle_count()`, `esp_cpu_set_cycle_count()` and `esp_cpu_dbgr_is_attached()` instead.
- The header file `esp_intr.h` has been deleted. Please include `esp_intr_alloc.h` to allocate and manipulate interrupts.
- The Panic API (functions/types/macros prefixed with `esp_panic`) has been made into a private API. Thus, the previous include path `#include "esp_panic.h"` has been removed. If users still require usage

of the Trax API (though this is not recommended), it can be included via `#include "esp_private/panic_reason.h"`. Besides, developers should include `esp_debug_helpers.h` instead to use any debug-related helper functions, e.g., print backtrace.
- The header file `soc_log.h` is now renamed to `esp_hw_log.h` and has been made private. Users are encouraged to use logging APIs provided under `esp_log.h` instead.
- The header files `spinlock.h`, `clk_ctrl_os.h`, and `rtc_wdt.h` must now be included without the `soc` prefix. For example, `#include "spinlock.h"`.
- `esp_chip_info()` returns the chip version in the format = 100 * `major eFuse version` + `minor eFuse version`. Thus, the `revision` in the `esp_chip_info_t` structure is expanded to uint16_t to fit the new format.

PSRAM

- The target-specific header file `spiram.h` and the header file `esp_spiram.h` have been removed. A new component `esp_psram` is created instead. For PSRAM/SPIRAM-related functions, users now include `esp_psram.h` and set the `esp_psram` component as a component requirement in their `CMakeLists.txt` project files.
- `esp_spiram_get_chip_size` and `esp_spiram_get_size` have been deleted. You should use `esp_psram_get_size` instead.

eFuse

- The parameter type of function `esp_secure_boot_read_key_digests()` changed from `ets_secure_boot_key_digests_t*` to `esp_secure_boot_key_digests_t*`. The new type is the same as the old one, except that the `allow_key_revoke` flag has been removed. The latter was always set to `true` in current code, not providing additional information.
- Added eFuse wafer revisions: major and minor. The `esp_efuse_get_chip_ver()` API is not compatible with these changes, so it was removed. Instead, please use the following APIs: `efuse_hal_get_major_chip_version()`, `efuse_hal_get_minor_chip_version()` or `efuse_hal_chip_revision()`.

esp_common `EXT_RAM_ATTR` is deprecated. Use the new macro `EXT_RAM_BSS_ATTR` to put .bss on PSRAM.

esp_system

- The header files `esp_random.h`, `esp_mac.h`, and `esp_chip_info.h`, which were all previously indirectly included via the header file `esp_system.h`, must now be included directly. These indirect inclusions from `esp_system.h` have been removed.
- The Backtrace Parser API (functions/types/macros prefixed with `esp_eh_frame_`) has been made into a private API. Thus, the previous include path `#include "eh_frame_parser.h"` has been removed. If users still require usage of the Backtrace Parser API (though this is not recommended), it can be included via `#include "esp_private/eh_frame_parser.h"`.
- The Interrupt Watchdog API (functions/types/macros prefixed with `esp_int_wdt_`) has been made into a private API. Thus, the previous include path `#include "esp_int_wdt.h"` has been removed. If users still require usage of the Interrupt Watchdog API (though this is not recommended), it can be included via `#include "esp_private/esp_int_wdt.h"`.

SOC Dependency

- Public API headers listed in the Doxyfiles will not expose unstable and unnecessary soc header files, such as `soc/soc.h` and `soc/rtc.h`. That means the user has to explicitly include them in their code if these "missing" header files are still wanted.
- Kconfig option `LEGACY_INCLUDE_COMMON_HEADERS` is also removed.
- The header file `soc/soc_memory_types.h` has been deprecated. Users should use the `esp_memory_utils.h` instead. Including `soc/soc_memory_types.h` will bring a build warning like `soc_memory_types.h is deprecated, please migrate to esp_memory_utils.h`

APP Trace One of the timestamp sources has changed from the legacy timer group driver to the new *GPTimer*. Kconfig choices like `APPTRACE_SV_TS_SOURCE_TIMER00` has been changed to `APPTRACE_SV_TS_SOURCE_GPTIMER`. User no longer need to choose the group and timer ID.

esp_timer The FRC2-based legacy implementation of esp_timer available on ESP32 has been removed. The simpler and more efficient implementation based on the LAC timer is now the only option.

ESP Image The image SPI speed enum definitions have been renamed.

- Enum `ESP_IMAGE_SPI_SPEED_80M` has been renamed to `ESP_IMAGE_SPI_SPEED_DIV_1`.
- Enum `ESP_IMAGE_SPI_SPEED_40M` has been renamed to `ESP_IMAGE_SPI_SPEED_DIV_2`.
- Enum `ESP_IMAGE_SPI_SPEED_26M` has been renamed to `ESP_IMAGE_SPI_SPEED_DIV_3`.
- Enum `ESP_IMAGE_SPI_SPEED_20M` has been renamed to `ESP_IMAGE_SPI_SPEED_DIV_4`.

Task Watchdog Timers

- The API for `esp_task_wdt_init()` has changed as follows:
  - Configuration is now passed as a configuration structure.
  - The function will now handle subscribing of the idle tasks if configured to do so.
- The former `CONFIG_ESP_TASK_WDT` configuration option has been renamed to *CONFIG_ESP_TASK_WDT_INIT* and a new *CONFIG_ESP_TASK_WDT_EN* option has been introduced.

FreeRTOS

Legacy API and Data Types Previously, the `configENABLE_BACKWARD_COMPATIBILITY` option was set by default, thus allowing pre FreeRTOS v8.0.0 function names and data types to be used. The `configENABLE_BACKWARD_COMPATIBILITY` is now disabled by default, thus legacy FreeRTOS names/types are no longer supportd by default. Users should do one of the followings:

- Update their code to remove usage of legacy FreeRTOS names/types.
- Enable the *CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY* to explicitly allow the usage of legacy names/types.

Tasks Snapshot The header `task_snapshot.h` has been removed from `freertos/task.h`. ESP-IDF developers should include `freertos/task_snapshot.h` if they need tasks snapshot API.

The function `vTaskGetSnapshot()` now returns `BaseType_t`. Return value shall be `pdTRUE` on success and `pdFALSE` otherwise.

FreeRTOS Asserts Previously, FreeRTOS asserts were configured separately from the rest of the system using the `FREERTOS_ASSERT` kconfig option. This option has now been removed and the configuration is now done through `COMPILER_OPTIMIZATION_ASSERTION_LEVEL`.

Port Macro API The file `portmacro_deprecated.h` which was added to maintain backward compatibility for deprecated APIs is removed. Users are advised to use the alternate functions for the deprecated APIs as listed below:

- `portENTER_CRITICAL_NESTED()` is removed. Users should use the `portSET_INTERRUPT_MASK_FROM_ISR()` macro instead.
- `portEXIT_CRITICAL_NESTED()` is removed. Users should use the `portCLEAR_INTERRUPT_MASK_FROM_ISR()` macro instead.
- `vPortCPUInitializeMutex()` is removed. Users should use the `spinlock_initialize()` function instead.
- `vPortCPUAcquireMutex()` is removed. Users should use the `spinlock_acquire()` function instead.

- `vPortCPUAcquireMutexTimeout()` is removed. Users should use the `spinlock_acquire()` function instead.
- `vPortCPUReleaseMutex()` is removed. Users should use the `spinlock_release()` function instead.

App Update

- The functions *`esp_ota_get_app_description()`* and *`esp_ota_get_app_elf_sha256()`* have been termed as deprecated. Please use the alternative functions *`esp_app_get_description()`* and *`esp_app_get_elf_sha256()`* respectively. These functions have now been moved to a new component esp_app_format. (Refer header file esp_app_desc.h)

Bootloader Support

- The *`esp_app_desc_t`* structure, which used to be declared in esp_app_format.h , is now declared in esp_app_desc.h .
- The function `bootloader_common_get_partition_description()` has now been made private. Please use the alternative function *`esp_ota_get_partition_description()`*. Note that this function takes *`esp_partition_t`* as its first argument instead of `esp_partition_pos_t`.

Chip Revision The bootloader checks the chip revision at the beginning of the application loading. The application can only be loaded if the version is >= *CONFIG_ESP32_REV_MIN* and < `CONFIG_ESP32_REV_MAX_FULL`.

During the OTA upgrade, the version requirements and chip revision in the application header are checked for compatibility. The application can only be updated if the version is >= *CONFIG_ESP32_REV_MIN* and < `CONFIG_ESP32_REV_MAX_FULL`.

Tools

IDF Monitor IDF Monitor makes the following changes regarding baud-rate:

- IDF monitor now uses the custom console baud-rate (*CONFIG_ESP_CONSOLE_UART_BAUDRATE*) by default instead of 115200.
- Setting a custom baud from menuconfig is no longer supported.
- A custom baud-rate can be specified from command line with the `idf.py monitor -b <baud>` command or through setting environment variables.
- Please note that the baud-rate argument has been renamed from `-B` to `-b` in order to be consistent with the global baud-rate `idf.py -b <baud>`. Run `idf.py monitor --help` for more information.

Deprecated Commands `idf.py` sub-commands and `cmake` target names have been unified to use hyphens (`-`) instead of underscores (_). Using a deprecated sub-command or target name will produce a warning. Users are advised to migrate to using the new sub-commands and target names. The following changes have been made:

Table 1: Deprecated Sub-command and Target Names

Old Name	New Name
efuse_common_table	efuse-common-table
efuse_custom_table	efuse-custom-table
erase_flash	erase-flash
partition_table	partition-table
partition_table-flash	partition-table-flash
post_debug	post-debug
show_efuse_table	show-efuse-table
erase_otadata	erase-otadata
read_otadata	read-otadata

Esptool The CONFIG_ESPTOOLPY_FLASHSIZE_DETECT option has been renamed to *CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE* and has been disabled by default. New and existing projects migrated to ESP-IDF v5.0 will have to set *CONFIG_ESPTOOLPY_FLASHSIZE*. If this is not possible due to an unknown flash size at build time, then *CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE* can be enabled. However, once enabled, to keep the digest valid, a SHA256 digest will no longer be appended to the image when updating the binary header with the flash size during flashing.

Windows Environment The Msys/Mingw-based Windows environment support got deprecated in ESP-IDF v4.0 and was entirely removed in v5.0. Please use *ESP-IDF Tools Installer* to set up a compatible environment. The options include Windows Command Line, Power Shell and the graphical user interface based on Eclipse IDE. In addition, a VS Code-based environment can be set up with the supported plugin: https://github.com/espressif/vscode-esp-idf-extension.

WiFi

WiFi Enterprise security APIs defined in *esp_wpa2.h* have been deprecated. Please use newer APIs from *esp_eap_client.h*.

Chapter 6

Libraries and Frameworks

6.1 Cloud Frameworks

ESP32 supports multiple cloud frameworks using agents built on top of ESP-IDF. Here are the pointers to various supported cloud frameworks' agents and examples:

6.1.1 ESP RainMaker

ESP RainMaker is a complete solution for accelerated AIoT development. ESP RainMaker on GitHub.

6.1.2 AWS IoT

https://github.com/espressif/esp-aws-iot is an open source repository for ESP32 based on Amazon Web Services' aws-iot-device-sdk-embedded-C.

6.1.3 Azure IoT

https://github.com/espressif/esp-azure is an open source repository for ESP32 based on Microsoft Azure' s azure-iot-sdk-c SDK.

6.1.4 Google IoT Core

https://github.com/espressif/esp-google-iot is an open source repository for ESP32 based on Google' s iot-device-sdk-embedded-c SDK.

6.1.5 Aliyun IoT

https://github.com/espressif/esp-aliyun is an open source repository for ESP32 based on Aliyun' s iotkit-embedded SDK.

6.1.6 Joylink IoT

https://github.com/espressif/esp-joylink is an open source repository for ESP32 based on Joylink' s joylink_dev_sdk SDK.

6.1.7 Tencent IoT

https://github.com/espressif/esp-welink is an open source repository for ESP32 based on Tencent's welink SDK.

6.1.8 Tencentyun IoT

https://github.com/espressif/esp-qcloud is an open source repository for ESP32 based on Tencentyun's qcloud-iot-sdk-embedded-c SDK.

6.1.9 Baidu IoT

https://github.com/espressif/esp-baidu-iot is an open source repository for ESP32 based on Baidu's iot-sdk-c SDK.

6.2 Espressif's Frameworks

Here you will find a collection of the official Espressif libraries and frameworks.

6.2.1 Espressif Audio Development Framework

The ESP-ADF is a comprehensive framework for audio applications including:

- CODEC's HAL
- Music Players and Recorders
- Audio Processing
- Bluetooth Speakers
- Internet Radios
- Hands-free devices
- Speech Recognition

This framework is available at GitHub: ESP-ADF.

6.2.2 ESP-CSI

ESP-CSI is an experimental implementation that uses the Wi-Fi Channel State Information to detect the presence of a human body.

See ESP-CSI project for more information about it.

6.2.3 Espressif DSP Library

The library provides algorithms optimized specifically for digital signal processing applications. This library supports:

- Matrix multiplication
- Dot product
- FFT (Fast Fourier Transform)
- IIR (Infinite Impulse Response)
- FIR (Finite Impulse Response)
- Vector math operations

This library is available here: ESP-DSP library.

6.2.4 ESP-WIFI-MESH Development Framework

This framework is based on the ESP-WIFI-MESH protocol with the following features:

- Fast network configuration
- Stable upgrade
- Efficient debugging
- LAN control
- Various application demos

ESP-MDF.

6.2.5 ESP-WHO

The ESP-WHO is a face detection and recognition framework using the ESP32 and camera. To know more about the project, see ESP-WHO on GitHub.

6.2.6 ESP RainMaker

ESP RainMaker is a complete solution for accelerated AIoT development. Using ESP RainMaker, you can create AIoT devices from the firmware to the integration with voice-assistant, phone apps and cloud backend.

ESP RainMaker on GitHub.

6.2.7 ESP-IoT-Solution

ESP-IoT-Solution contains commonly used device drivers and code frameworks when developing IoT systems. The device drivers and code frameworks within the ESP-IoT-Solution are organized as separate components, allowing them to be easily integrated into an ESP-IDF project.

ESP-IoT-Solution includes:

- Device drivers for sensors, display, audio, GUI, input, actuators, etc.
- Framework and documentation for low power, security, storage, etc.
- Guide for Espressif open source solutions from practical application point.

ESP-IoT-Solution on GitHub.

6.2.8 ESP-Protocols

ESP-Protocols repository contains collection of protocol components for ESP-IDF. The code within the ESP-Protocols is organized into separate components, allowing them to be easily integrated into an ESP-IDF project. In addition to that, each component is available in IDF Component Registry.

ESP-Protocols components:

- esp_modem enables connectivity with GSM/LTE modems using AT commands or PPP protocol, see the esp_modem documentation.
- mdns (mDNS) is a multicast UDP service that is used to provide local network service and host discovery, see the mdns documentation.
- esp_websocket_client is a managed component for *esp-idf* that contains implementation of [WebSocket protocol client](https://datatracker.ietf.org/doc/html/rfc6455) for ESP32, see the esp_websocket_client documentation.
- asio is a cross-platform C++ library, see https://think-async.com/Asio/. It provides a consistent asynchronous model using a modern C++ approach. , see the asio documentation.

6.2.9 ESP-BSP

ESP-BSP repository contains Board Support Packages (BSPs) for various Espressif's and 3rd party development boards. BSPs are useful for quick start on a supported board. Usually they contain pinout definition and helper functions, that will initialize peripherals for the specific board. Additionally, the BSP would contain drivers for external chips populated on the development board, such as sensors, displays, audio codecs etc.

Chapter 7

Contributions Guide

We welcome contributions to the esp-idf project!

7.1 How to Contribute

Contributions to esp-idf - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via Github Pull Requests.

7.2 Before Contributing

Before sending us a Pull Request, please consider this list of points:

- Is the contribution entirely your own work, or already licensed under an Apache License 2.0 compatible Open Source License? If not then we unfortunately cannot accept it. Please check the *Copyright Header Guide* for additional information.
- Does any new code conform to the esp-idf *Style Guide*?
- Have you installed the *pre-commit hook* for esp-idf project?
- Does the code documentation follow requirements in *Documenting Code*?
- Is the code adequately commented for people to understand how it is structured?
- Is there documentation or examples that go with code contributions? There are additional suggestions for writing good examples in examples readme.
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- Example contributions are also welcome. Please check the *Creating Examples* guide for these.
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like "fixed typo" squashed into previous commits?
- If you're unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

7.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for in-house automated testing.

If this process passes, it will be merged into the public GitHub repository.

7.4 Legal Part

Before a contribution can be accepted, you will need to sign our *Contributor Agreement*. You will be prompted for this automatically as part of the Pull Request process.

7.5 Related Documents

7.5.1 Espressif IoT Development Framework Style Guide

About This Guide

Purpose of this style guide is to encourage use of common coding practices within the ESP-IDF.

Style guide is a set of rules which are aimed to help create readable, maintainable, and robust code. By writing code which looks the same way across the code base we help others read and comprehend the code. By using same conventions for spaces and newlines we reduce chances that future changes will produce huge unreadable diffs. By following common patterns for module structure and by using language features consistently we help others understand code behavior.

We try to keep rules simple enough, which means that they can not cover all potential cases. In some cases one has to bend these simple rules to achieve readability, maintainability, or robustness.

When doing modifications to third-party code used in ESP-IDF, follow the way that particular project is written. That will help propose useful changes for merging into upstream project.

C Code Formatting

Naming

- Any variable or function which is only used in a single source file should be declared `static`.
- Public names (non-static variables and functions) should be namespaced with a per-component or per-unit prefix, to avoid naming collisions. ie `esp_vfs_register()` or `esp_console_run()`. Starting the prefix with `esp_` for Espressif-specific names is optional, but should be consistent with any other names in the same component.
- Static variables should be prefixed with `s_` for easy identification. For example, `static bool s_invert`.
- Avoid unnecessary abbreviations (ie shortening `data` to `dat`), unless the resulting name would otherwise be very long.

Indentation Use 4 spaces for each indentation level. Don't use tabs for indentation. Configure the editor to emit 4 spaces each time you press tab key.

Vertical Space Place one empty line between functions. Don't begin or end a function with an empty line.

```
void function1()
{
    do_one_thing();
    do_another_thing();
                                    // INCORRECT, don't place empty line here
}
                                    // place empty line here
void function2()
{
                                    // INCORRECT, don't use an empty line here
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The maximum line length is 120 characters as long as it doesn't seriously affect the readability.

Horizontal Space Always add single space after conditional and loop keywords:

```
if (condition) {     // correct
    // ...
}

switch (n) {         // correct
    case 0:
        // ...
}

for(int i = 0; i < CONST; ++i) {     // INCORRECT
    // ...
}
```

Add single space around binary operators. No space is necessary for unary operators. It is okay to drop space around multiply and divide operators:

```
const int y = y0 + (x - x0) * (y1 - y0) / (x1 - x0);     // correct

const int y = y0 + (x - x0)*(y1 - y0)/(x1 - x0);         // also okay

int y_cur = -y;                                          // correct
++y_cur;

const int y = y0+(x-x0)*(y1-y0)/(x1-x0);                 // INCORRECT
```

No space is necessary around `.` and `->` operators.

Sometimes adding horizontal space within a line can help make code more readable. For example, you can add space to align function arguments:

```
esp_rom_gpio_connect_in_signal(PIN_CAM_D6,   I2S0I_DATA_IN14_IDX, false);
esp_rom_gpio_connect_in_signal(PIN_CAM_D7,   I2S0I_DATA_IN15_IDX, false);
esp_rom_gpio_connect_in_signal(PIN_CAM_HREF, I2S0I_H_ENABLE_IDX,  false);
esp_rom_gpio_connect_in_signal(PIN_CAM_PCLK, I2S0I_DATA_IN15_IDX, false);
```

Note however that if someone goes to add new line with a longer identifier as first argument (e.g. `PIN_CAM_VSYNC`), it will not fit. So other lines would have to be realigned, adding meaningless changes to the commit.

Therefore, use horizontal alignment sparingly, especially if you expect new lines to be added to the list later.

Never use TAB characters for horizontal alignment.

Never add trailing whitespace at the end of the line.

Braces

- Function definition should have a brace on a separate line:

```
// This is correct:
void function(int arg)
{

}
```

(continues on next page)

(continued from previous page)

```
// NOT like this:
void function(int arg) {

}
```

- Within a function, place opening brace on the same line with conditional and loop statements:

```
if (condition) {
    do_one();
} else if (other_condition) {
    do_two();
}
```

Comments Use `//` for single line comments. For multi-line comments it is okay to use either `//` on each line or a `/* */` block.

Although not directly related to formatting, here are a few notes about using comments effectively.

- Don't use single comments to disable some functionality:

```
void init_something()
{
    setup_dma();
    // load_resources();                // WHY is this thing commented, asks␣
↪the reader?
    start_timer();
}
```

- If some code is no longer required, remove it completely. If you need it you can always look it up in git history of this file. If you disable some call because of temporary reasons, with an intention to restore it in the future, add explanation on the adjacent line:

```
void init_something()
{
    setup_dma();
    // TODO: we should load resources here, but loader is not fully integrated␣
↪yet.
    // load_resources();
    start_timer();
}
```

- Same goes for `#if 0 ... #endif` blocks. Remove code block completely if it is not used. Otherwise, add comment explaining why the block is disabled. Don't use `#if 0 ... #endif` or comments to store code snippets which you may need in the future.
- Don't add trivial comments about authorship and change date. You can always look up who modified any given line using git. E.g. this comment adds clutter to the code without adding any useful information:

```
void init_something()
{
    setup_dma();
    // XXX add 2016-09-01
    init_dma_list();
    fill_dma_item(0);
    // end XXX add
    start_timer();
}
```

Line Endings Commits should only contain files with LF (Unix style) endings.

Windows users can configure git to check out CRLF (Windows style) endings locally and commit LF endings by setting the `core.autocrlf` setting. *Github has a document about setting this option <github-line-endings>.*

If you accidentally have some commits in your branch that add LF endings, you can convert them to Unix by running this command in an MSYS2 or Unix terminal (change directory to the IDF working directory and check the correct branch is currently checked out, beforehand):

```
git rebase --exec 'git diff-tree --no-commit-id --name-only -r HEAD | xargs␣
↪dos2unix && git commit -a --amend --no-edit --allow-empty' master
```

(Note that this line rebases on master, change the branch name at the end to rebase on another branch.)

For updating a single commit, it's possible to run `dos2unix FILENAME` and then run `git commit --amend`

Formatting Your Code You can use `astyle` program to format your code according to the above recommendations.

If you are writing a file from scratch, or doing a complete rewrite, feel free to re-format the entire file. If you are changing a small portion of file, don't re-format the code you didn't change. This will help others when they review your changes.

To re-format a file, run:

```
tools/format.sh components/my_component/file.c
```

Type Definitions Should be snake_case, ending with _t suffix:

```
typedef int signed_32_bit_t;
```

Enum Enums should be defined through the *typedef* and be namespaced:

```
typedef enum
{
    MODULE_FOO_ONE,
    MODULE_FOO_TWO,
    MODULE_FOO_THREE
} module_foo_t;
```

Assertions The standard C `assert()` function, defined in `assert.h` should be used to check conditions that should be true in source code. In the default configuration, an assert condition that returns `false` or 0 will call `abort()` and trigger a *Fatal Error*.

`assert()` should only be used to detect unrecoverable errors due to a serious internal logic bug or corruption, where it's not possible for the program to continue. For recoverable errors, including errors that are possible due to invalid external input, an *error value should be returned*.

Note: When asserting a value of type `esp_err_t``is equal to ``ESP_OK`, use the *ESP_ERROR_CHECK macro* instead of an `assert()`.

It's possible to configure ESP-IDF projects with assertions disabled (see *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*). Therefore, functions called in an `assert()` statement should not have side-effects.

It's also necessary to use particular techniques to avoid "variable set but not used" warnings when assertions are disabled, due to code patterns such as:

```
int res = do_something();
assert(res == 0);
```

Once the `assert` is optimized out, the `res` value is unused and the compiler will warn about this. However the function `do_something()` must still be called, even if assertions are disabled.

When the variable is declared and initialized in a single statement, a good strategy is to cast it to `void` on a new line. The compiler will not produce a warning, and the variable can still be optimized out of the final binary:

```
int res = do_something();
assert(res == 0);
(void)res;
```

If the variable is declared separately, for example if it is used for multiple assertions, then it can be declared with the GCC attribute `__attribute__((unused))`. The compiler will not produce any unused variable warnings, but the variable can still be optimized out:

```
int res __attribute__((unused));

res = do_something();
assert(res == 0);

res = do_something_else();
assert(res != 0);
```

Header file guards

All public facing header files should have preprocessor guards. A pragma is preferred:

```
#pragma once
```

over the following pattern:

```
#ifndef FILE_NAME_H
#define FILE_NAME_H
...
#endif // FILE_NAME_H
```

In addition to guard macros, all C header files should have `extern "C"` guards to allow the header to be used from C++ code. Note that the following order should be used: `pragma once`, then any `#include` statements, then `extern "C"` guards:

```
#pragma once

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* declarations go here */

#ifdef __cplusplus
}
#endif
```

Include statements

When writing `#include` statements, try to maintain the following order:

- C standard library headers.
- Other POSIX standard headers and common extensions to them (such as `sys/queue.h`.)
- Common IDF headers (`esp_log.h`, `esp_system.h`, `esp_timer.h`, `esp_sleep.h`, etc.)

- Headers of other components, such as FreeRTOS.
- Public headers of the current component.
- Private headers.

Use angle brackets for C standard library headers and other POSIX headers (`#include <stdio.h>`).

Use double quotes for all other headers (`#include "esp_log.h"`).

C++ Code Formatting

The same rules as for C apply. Where they are not enough, apply the following rules.

File Naming C++ Header files have the extension `.hpp`. C++ source files have the extension `.cpp`. The latter is important for the compiler to distinguish them from normal C source files.

Naming

- **Class and struct** names shall be written in `CamelCase` with a capital letter as beginning. Member variables and methods shall be in `snake_case`.
- **Namespaces** shall be in lower `snake_case`.
- **Templates** are specified in the line above the function declaration.
- Interfaces in terms of Object-Oriented Programming shall be named without the suffix `...Interface`. Later, this makes it easier to extract interfaces from normal classes and vice versa without making a breaking change.

Member Order in Classes In order of precedence:

- First put the public members, then the protected, then private ones. Omit public, protected or private sections without any members.
- First put constructors/destructors, then member functions, then member variables.

For example:

```
class ForExample {
public:
    // first constructors, then default constructor, then destructor
    ForExample(double example_factor_arg);
    ForExample();
    ~ForExample();

    // then remaining pubic methods
    set_example_factor(double example_factor_arg);

    // then public member variables
    uint32_t public_data_member;

private:
    // first private methods
    void internal_method();

    // then private member variables
    double example_factor;
};
```

Spacing

- Don't indent inside namespaces.
- Put `public`, `protected` and `private` labels at the same indentation level as the corresponding `class` label.

Simple Example

```
// file spaceship.h
#ifndef SPACESHIP_H_
#define SPACESHIP_H_
#include <cstdlib>

namespace spaceships {

class SpaceShip {
public:
    SpaceShip(size_t crew);
    size_t get_crew_size() const;

private:
    const size_t crew;
};

class SpaceShuttle : public SpaceShip {
public:
    SpaceShuttle();
};

class Sojuz : public SpaceShip {
public:
    Sojuz();
};

template <typename T>
class CargoShip {
public:
    CargoShip(const T &cargo);

private:
    T cargo;
};

} // namespace spaceships

#endif // SPACESHIP_H_

// file spaceship.cpp
#include "spaceship.h"

namespace spaceships {

// Putting the curly braces in the same line for constructors is OK if it only
↪initializes
// values in the initializer list
SpaceShip::SpaceShip(size_t crew) : crew(crew) { }

size_t SpaceShip::get_crew_size() const
{
    return crew;
}

SpaceShuttle::SpaceShuttle() : SpaceShip(7)
{
    // doing further initialization
}

Sojuz::Sojuz() : SpaceShip(3)
{
```

(continues on next page)

(continued from previous page)

```
    // doing further initialization
}

template <typename T>
CargoShip<T>::CargoShip(const T &cargo) : cargo(cargo) { }

} // namespace spaceships
```

CMake Code Style

- Indent with four spaces.
- Maximum line length 120 characters. When splitting lines, try to focus on readability where possible (for example, by pairing up keyword/argument pairs on individual lines).
- Don't put anything in the optional parentheses after `endforeach()`, `endif()`, etc.
- Use lowercase (`with_underscores`) for command, function, and macro names.
- For locally scoped variables, use lowercase (`with_underscores`).
- For globally scoped variables, use uppercase (`WITH_UNDERSCORES`).
- Otherwise follow the defaults of the cmake-lint project.

Configuring the Code Style for a Project Using EditorConfig

EditorConfig helps developers define and maintain consistent coding styles between different editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

For more information, see EditorConfig Website.

Documenting Code

Please see the guide here: *Documenting Code*.

Structure

To be written.

Language Features

To be written.

7.5.2 Install pre-commit Hook for ESP-IDF Project

Required Dependency

Python 3.7.* or above. This is our recommended python version for IDF developers.

If you still have python versions not compatible, update your python versions before installing the pre-commit hook.

Install pre-commit

Run `pip install pre-commit`

Install pre-commit hook

1. Go to the IDF Project Directory
2. Run `pre-commit install --allow-missing-config`. Install hook by this approach will let you commit successfully even in branches without the `.pre-commit-config.yaml`
3. pre-commit hook will run automatically when you're running `git commit` command

Uninstall pre-commit

Run `pre-commit uninstall`

What's More?

For detailed usage, please refer to the documentation of pre-commit.

Common Problems For Windows Users

`/usr/bin/env: python: Permission denied.`

If you're in Git Bash, please check the python executable location by run `which python`.

If the executable is under `~/AppData/Local/Microsoft/WindowsApps/`, then it's a link to Windows AppStore, not a real one.

Please install python manually and update this in your `PATH` environment variable.

Your %USERPROFILE% contains non-ASCII characters

`pre-commit` may fail when initializing an environment for a particular hook when the path of `pre-commit`'s cache contains non-ASCII characters. The solution is to set `PRE_COMMIT_HOME` to a path containing only standard characters before running pre-commit.

- CMD: `set PRE_COMMIT_HOME=C:\somepath\pre-commit`
- PowerShell: `$Env:PRE_COMMIT_HOME = "C:\somepath\pre-commit"`
- git bash: `export PRE_COMMIT_HOME="/c/somepath/pre-commit"`

7.5.3 Documenting Code

The purpose of this description is to provide quick summary on documentation style used in espressif/esp-idf repository and how to add new documentation.

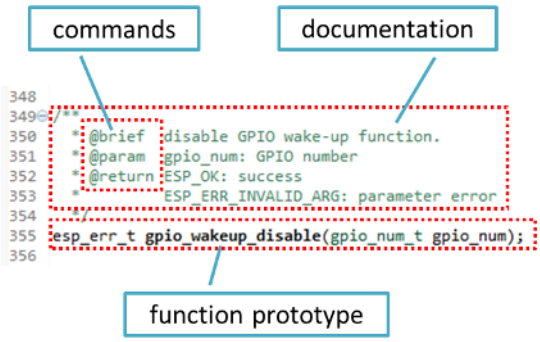
Introduction

When documenting code for this repository, please follow Doxygen style. You are doing it by inserting special commands, for instance `@param`, into standard comments blocks, for example:

```
/**
 * @param ratio this is oxygen to air ratio
 */
```

Doxygen is phrasing the code, extracting the commands together with subsequent text, and building documentation out of it.

Typical comment block, that contains documentation of a function, looks like below.

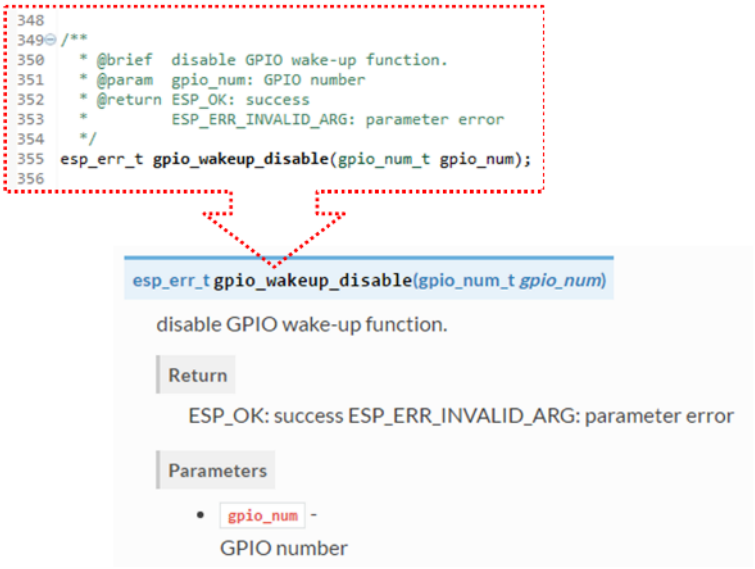


Doxygen supports couple of formatting styles. It also gives you great flexibility on level of details to include in documentation. To get familiar with available features, please check data rich and very well organized Doxygen Manual.

Why we need it?

The ultimate goal is to ensure that all the code is consistently documented, so we can use tools like Sphinx and Breathe to aid preparation and automatic updates of API documentation when the code changes.

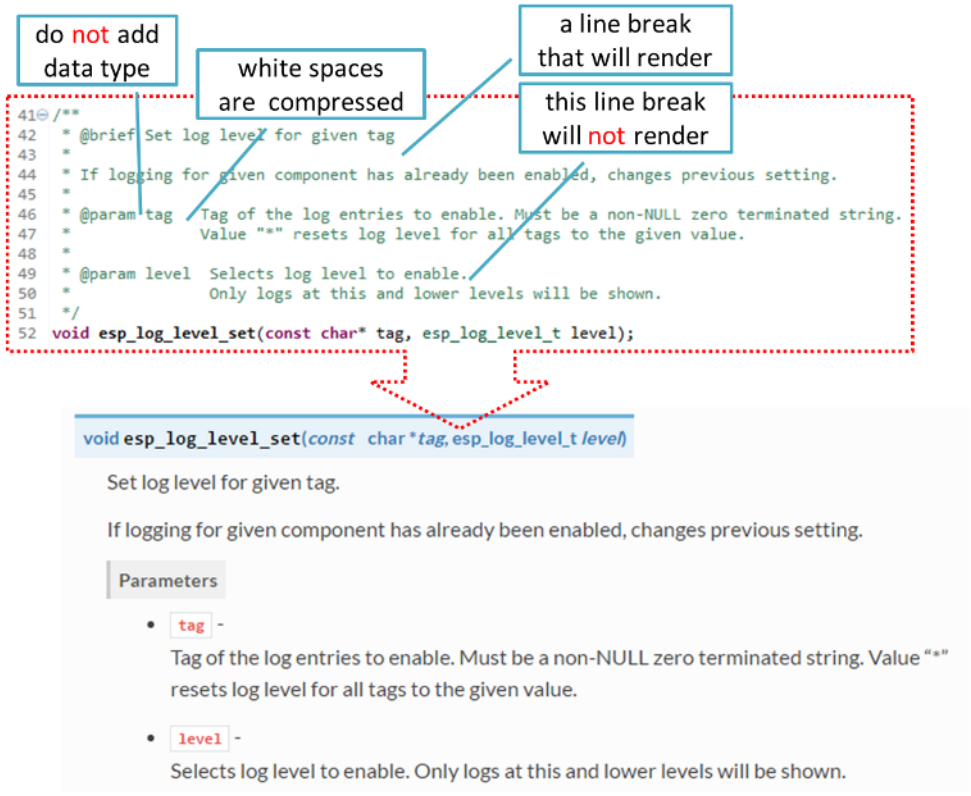
With these tools the above piece of code renders like below:



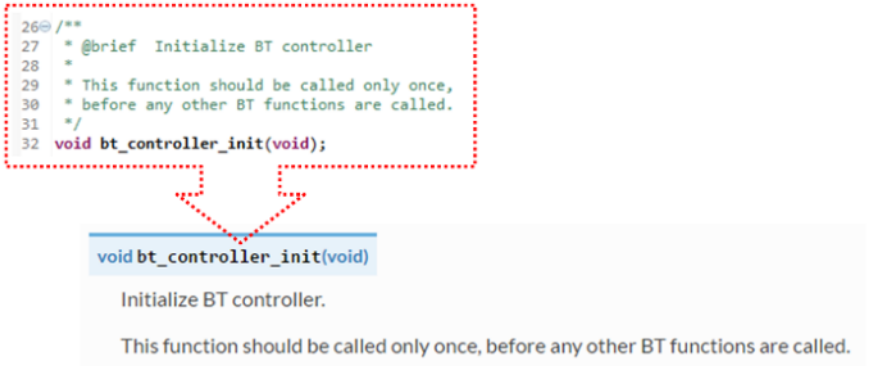
Go for it!

When writing code for this repository, please follow guidelines below.

1. Document all building blocks of code: functions, structs, typedefs, enums, macros, etc. Provide enough information about purpose, functionality and limitations of documented items, as you would like to see them documented when reading the code by others.
2. Documentation of function should describe what this function does. If it accepts input parameters and returns some value, all of them should be explained.
3. Do not add a data type before parameter or any other characters besides spaces. All spaces and line breaks are compressed into a single space. If you like to break a line, then break it twice.



4. If function has void input or does not return any value, then skip `@param` or `@return`



5. When documenting a `define` as well as members of a `struct` or `enum`, place specific comment like below after each member.

6. To provide well formatted lists, break the line after command (like `@return` in example below).

```
*
* @return
*     - ESP_OK if erase operation was successful
```

(continues on next page)

(continued from previous page)

```
*     - ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
*     - ESP_ERR_NVS_READ_ONLY if handle was opened as read only
*     - ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
*     - other error codes from the underlying storage driver
*
```

7. Overview of functionality of documented header file, or group of files that make a library, should be placed in a separate `README.rst` file of the same directory. If this directory contains header files for different APIs, then the file name should be `apiname-readme.rst`.

Go one extra mile

Here are a couple of tips on how you can make your documentation even better and more useful to the reader and writer.

When writing codes, please follow the guidelines below:

1. Add code snippets to illustrate implementation. To do so, enclose snippet using `@code{c}` and `@endcode` commands.

```
*
* @code{c}
* // Example of using nvs_get_i32:
* int32_t max_buffer_size = 4096; // default value
* esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
* assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
* // if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
* // have its default value.
* @endcode
*
```

   The code snippet should be enclosed in a comment block of the function that it illustrates.
2. To highlight some important information use command `@attention` or `@note`.

```
*
* @attention
*     1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode
*     2. If the ESP32 is connected to an AP, call esp_wifi_disconnect to␣
↪disconnect.
*
```

   Above example also shows how to use a numbered list.
3. To provide common description to a group of similar functions, enclose them using `/**@{*/` and `/**@}*/` markup commands:

```
/**@{*/
/**
 * @brief common description of similar functions
 *
 */
void  first_similar_function (void);
void second_similar_function (void);
/**@}*/
```

   For practical example see nvs_flash/include/nvs.h.
4. You may want to go even further and skip some code like repetitive defines or enumerations. In such case, enclose the code within `/** @cond */` and `/** @endcond */` commands. Example of such implementation is provided in driver/include/driver/gpio.h.
5. Use markdown to make your documentation even more readable. You will add headers, links, tables and more.

```
*
* [ESP32 Technical Reference Manual](https://www.espressif.com/sites/default/
↪files/documentation/esp32_technical_reference_manual_en.pdf)
*
```

Note: Code snippets, notes, links, etc. will not make it to the documentation, if not enclosed in a comment block associated with one of documented objects.

6. Prepare one or more complete code examples together with description. Place description to a separate file `README.md` in specific folder of examples directory.

Standardize Document Format

When it comes to text, please follow guidelines below to provide well formatted Markdown (.md) or reST (.rst) documents.

1. Please ensure that one paragraph is written in one line. Don't break lines like below. Breaking lines to enhance readability is only suitable for writing codes. To make the text easier to read, it is recommended to place an empty line to separate the paragraph.

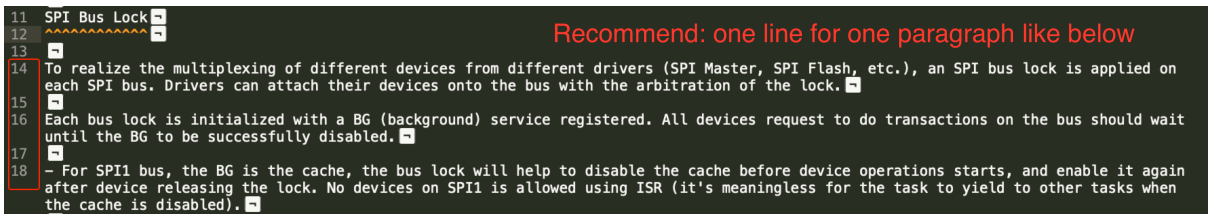


Fig. 1: One line for one paragraph (click to enlarge)

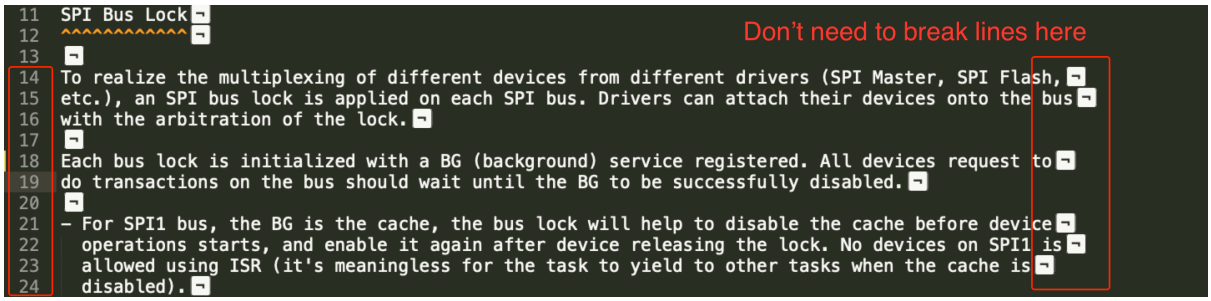


Fig. 2: No line breaks within the same paragraph (click to enlarge)

2. Please make the line number of CN and EN documents consistent like below. The benefit of this approach is that it can save time for both writers and translators. When non-bilingual writers need to update text, they only need to update the same line in the corresponding CN or EN document. For translators, if documents are updated in English, then translators can quickly locate where to update in the corresponding CN document later. Besides, by comparing the total number of lines in EN and CN documents, you can quickly find out whether the CN version lags behind the EN version.

Building Documentation

The documentation is built with the *esp-docs* Python package, which is a wrapper around Sphinx

To install it simply do:

```
1  ********************************
2  Getting Started with VS Code IDE
3  ********************************
4
5  :link_to_translation:`zh_CN:[中文]`
6
7  We have official support for VS Code and we aim to provide complete
   end to end support for all actions related to ESP-IDF namely build,
   flash, monitor, debug, tracing, core-dump, System Trace Viewer, etc.
8
9  Quick Install Guide
10 ==================
11
12 Recommended way to install ESP-IDF Visual Studio Code Extension is by
   downloading it from `VS Code Marketplace <https://marketplace.
   visualstudio.com/items?itemName=espressif.esp-idf-extension>`_ or
   following `Quick Installation Guide <https://github.com/espressif/
   vscode-esp-idf-extension/blob/master/docs/tutorial/install.md>`_.
13
14 Review the `tutorials <https://github.com/espressif/
   vscode-esp-idf-extension/blob/master/docs/tutorial/toc.md>` for
   ESP-IDF Visual Studio Code Extension to learn how to use all features.
15
16 Supported Features
17 ================
```

```
1  ********************************
2  VS Code IDE 快速入门
3  ********************************
4
5  :link_to_translation:`en:[English]`
6
7  我们支持 VS code，并且致力于为所有与 ESP-IDF 相关的操作提供完善的端到端支持，包
   括构建、烧录、监控、调试、追踪、core-dump、以及系统追踪查看器等操作。
8
9  快速安装指南
10 ==========
11
12 推荐您从 `VS Code 插件市场 <https://marketplace.visualstudio.com/items?
   itemName=espressif.esp-idf-extension>`_ 中下载 ESP-IDF VS Code 插件，或
   根据 `快速安装指南 <https://github.com/espressif/
   vscode-esp-idf-extension/blob/master/docs/tutorial/install.md>`_ 安装
   ESP-IDF VS Code 插件。
13
14 查看 ESP-IDF VS Code 插件 `教程 <https://github.com/espressif/
   vscode-esp-idf-extension/blob/master/docs/tutorial/toc.md>` 了解如何使用
   所有功能。
15
16 支持如下功能
17 ==========
18
```

Fig. 3: Keep the line number for EN and CN documents consistent (click to enlarge)

```
pip install esp-docs
```

After a successful install then the documentation can be built from the docs folder with:

```
build-docs build
```

or for specific target and language with:

```
build-docs -t esp32 -l en build
```

For more in-depth documentation about *esp-docs* features please see the documentation in the esp-docs repository.

Wrap up

We love good code that is doing cool things. We love it even better, if it is well documented, so we can quickly make it run and also do the cool things.

Go ahead, contribute your code and documentation!

Related Documents

- *API Documentation Template*

7.5.4 Creating Examples

Each ESP-IDF example is a complete project that someone else can copy and adapt the code to solve their own problem. Examples should demonstrate ESP-IDF functionality, while keeping this purpose in mind.

Structure

- The `main` directory should contain a source file named `(something)_example_main.c` with the main functionality.
- If the example has additional functionality, split it logically into separate C or C++ source files under `main` and place a corresponding header file in the same directory.

- If the example has a lot of additional functionality, consider adding a `components` directory to the example project and make some example-specific components with library functionality. Only do this if the components are specific to the example, if they're generic or common functionality then they should be added to ESP-IDF itself.
- The example should have a `README.md` file. Use the template example README and adapt it for your particular example.
- Examples should have a `pytest_<example name>.py` file for running an automated example test. If submitting a GitHub Pull Request which includes an example, it's OK not to include this file initially. The details can be discussed as part of the Pull Request. Please refer to *IDF Tests with Pytest Guide* for details.

General Guidelines

Example code should follow the *Espressif IoT Development Framework Style Guide*.

Checklist

Checklist before submitting a new example:

- Example project name (in `README.md`) uses the word "example". Use "example" instead of "demo", "test" or similar words.
- Example does one distinct thing. If the example does more than one thing at a time, split it into two or more examples.
- Example has a `README.md` file which is similar to the template example README .
- Functions and variables in the example are named according to *naming section of the style guide*. (For non-static names which are only specific to the example's source files, you can use `example` or something similar as a prefix.)
- All code in the example is well structured and commented.
- Any unnecessary code (old debugging logs, commented-out code, etc.) is removed from the example.
- Options in the example (like network names, addresses, etc) are not hard-coded. Use configuration items if possible, or otherwise declare macros or constants)
- Configuration items are provided in a `KConfig.projbuild` file with a menu named "Example Configuration". See existing example projects to see how this is done.
- All original example code has a license header saying it is "in the public domain / CC0", and a warranty disclaimer clause. Alternatively, the example is licensed under Apache License 2.0. See existing examples for headers to adapt from.
- Any adapted or third party example code has the original license header on it. This code must be licensed compatible with Apache License 2.0.

7.5.5 API Documentation Template

Note: *INSTRUCTIONS*

1. Use this file (docs/en/api-reference/template.rst) as a template to document API.
2. Change the file name to the name of the header file that represents documented API.
3. Include respective files with descriptions from the API folder using `..include::`
   - README.rst
   - example.rst
   - ...
4. Optionally provide description right in this file.
5. Once done, remove all instructions like this one and any superfluous headers.

Overview

Note: *INSTRUCTIONS*

1. Provide overview where and how this API may be used.
2. Where applicable include code snippets to illustrate functionality of particular functions.
3. To distinguish between sections, use the following heading levels:
   - `#` with overline, for parts
   - `*` with overline, for chapters
   - `=`, for sections
   - `-`, for subsections
   - `^`, for subsubsections
   - `"`, for paragraphs

Application Example

Note: *INSTRUCTIONS*

1. Prepare one or more practical examples to demonstrate functionality of this API.
2. Each example should follow pattern of projects located in `esp-idf/examples/` folder.
3. Place example in this folder complete with `README.md` file.
4. Provide overview of demonstrated functionality in `README.md`.
5. With good overview reader should be able to understand what example does without opening the source code.
6. Depending on complexity of example, break down description of code into parts and provide overview of functionality of each part.
7. Include flow diagram and screenshots of application output if applicable.
8. Finally add in this section synopsis of each example together with link to respective folder in `esp-idf/examples/`.

API Reference

Note: *INSTRUCTIONS*

1. This repository provides for automatic update of API reference documentation using *code markup retrieved by Doxygen from header files*.

1. Update is done on each documentation build by invoking Sphinx extension *:esp_extensions/run_doxygen.py* for all header files listed in the `INPUT` statement of docs/doxygen/Doxyfile.

1. Each line of the `INPUT` statement (other than a comment that begins with `##`) contains a path to header file `*.h` that will be used to generate corresponding `*.inc` files:

```
##
## Wi-Fi - API Reference
##
../components/esp32/include/esp_wifi.h \
../components/esp32/include/esp_smartconfig.h \
```

1. When the headers are expanded, any macros defined by default in `sdkconfig.h` as well as any macros defined in SOC-specific `include/soc/*_caps.h` headers will be expanded. This allows the headers to include/exclude material based on the `IDF_TARGET` value.

1. The `*.inc` files contain formatted reference of API members generated automatically on each documentation build. All `*.inc` files are placed in Sphinx `_build` directory. To see directives generated for e.g. `esp_wifi.h`, run `python gen-dxd.py esp32/include/esp_wifi.h`.

1. To show contents of `*.inc` file in documentation, include it as follows:

```
.. include-build-file:: inc/esp_wifi.inc
```

   For example see docs/en/api-reference/network/esp_wifi.rst

1. Optionally, rather that using `*.inc` files, you may want to describe API in you own way. See docs/en/api-reference/storage/fatfs.rst for example.
   Below is the list of common `.. doxygen...::` directives:
   - Functions - `.. doxygenfunction:: name_of_function`
   - Unions - `.. doxygenunion:: name_of_union`
   - Structures - `.. doxygenstruct:: name_of_structure` together with `:members:`
   - Macros - `.. doxygendefine:: name_of_define`
   - Type Definitions - `.. doxygentypedef:: name_of_type`
   - Enumerations - `.. doxygenenum:: name_of_enumeration`

   See Breathe documentation for additional information.
   To provide a link to header file, use the *link custom role* directive as follows:

```
* :component_file:`path_to/header_file.h`
```

1. In any case, to generate API reference, the file docs/doxygen/Doxyfile should be updated with paths to `*.h` headers that are being documented.

1. When changes are committed and documentation is build, check how this section has been rendered. *Correct annotations* in respective header files, if required.

7.5.6 Contributor Agreement

Individual Contributor Non-Exclusive License Agreement including the Traditional Patent License OPTION

Thank you for your interest in contributing to this Espressif project hosted on GitHub ("We" or "Us").

The purpose of this contributor agreement ("Agreement") is to clarify and document the rights granted by contributors to Us. To make this document effective, please follow the instructions in the *Contributions Guide*.

1. DEFINITIONS **"You"** means the Individual Copyright owner who submits a Contribution to Us. If You are an employee and submit the Contribution as part of your employment, You have had Your employer approve this Agreement or sign the Entity version of this document.

"Contribution" means any original work of authorship (software and/or documentation) including any modifications or additions to an existing work, Submitted by You to Us, in which You own the Copyright. If You do not own the Copyright in the entire work of authorship, please contact Us by submitting a comment on GitHub.

"Copyright" means all rights protecting works of authorship owned or controlled by You, including copyright, moral and neighboring rights, as appropriate, for the full term of their existence including any extensions by You.

"Material" means the software or documentation made available by Us to third parties. When this Agreement covers more than one software project, the Material means the software or documentation to which the Contribution was Submitted. After You Submit the Contribution, it may be included in the Material.

"Submit" means any form of physical, electronic, or written communication sent to Us, including but not limited to electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, Us, but excluding communication that is conspicuously marked or otherwise designated in writing by You as "Not a Contribution."

"Submission Date" means the date You Submit a Contribution to Us.

"Documentation" means any non-software portion of a Contribution.

2. LICENSE GRANT 2.1 Copyright License to Us

Subject to the terms and conditions of this Agreement, You hereby grant to Us a worldwide, royalty-free, NON-exclusive, perpetual and irrevocable license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, under the Copyright covering the Contribution to use the Contribution by all means, including, but not limited to:

- to publish the Contribution,
- to modify the Contribution, to prepare derivative works based upon or containing the Contribution and to combine the Contribution with other software code,
- to reproduce the Contribution in original or modified form,
- to distribute, to make the Contribution available to the public, display and publicly perform the Contribution in original or modified form.

2.2 Moral Rights remain unaffected to the extent they are recognized and not waivable by applicable law. Notwithstanding, You may add your name in the header of the source code files of Your Contribution and We will respect this attribution when using Your Contribution.

3. PATENTS 3.1 Patent License

Subject to the terms and conditions of this Agreement You hereby grant to us a worldwide, royalty-free, non-exclusive, perpetual and irrevocable (except as stated in Section 3.2) patent license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, to make, have made, use, sell, offer for sale, import and otherwise transfer the Contribution and the Contribution in combination with the Material (and portions of such combination). This license applies to all patents owned or controlled by You, whether already acquired or hereafter acquired, that would be infringed by making, having made, using, selling, offering for sale, importing or otherwise transferring of Your Contribution(s) alone or by combination of Your Contribution(s) with the Material.

3.2 Revocation of Patent License

You reserve the right to revoke the patent license stated in section 3.1 if we make any infringement claim that is targeted at your Contribution and not asserted for a Defensive Purpose. An assertion of claims of the Patents shall be considered for a "Defensive Purpose" if the claims are asserted against an entity that has filed, maintained, threatened, or voluntarily participated in a patent infringement lawsuit against Us or any of Our licensees.

4. DISCLAIMER THE CONTRIBUTION IS PROVIDED "AS IS". MORE PARTICULARLY, ALL EXPRESS OR IMPLIED WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED BY YOU TO US AND BY US TO YOU. TO THE EXTENT THAT ANY SUCH WARRANTIES CANNOT BE DISCLAIMED, SUCH WARRANTY IS LIMITED IN DURATION TO THE MINIMUM PERIOD PERMITTED BY LAW.

5. Consequential Damage Waiver TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL YOU OR US BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF ANTICIPATED SAVINGS, LOSS OF DATA, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL AND EXEMPLARY DAMAGES ARISING OUT OF THIS AGREEMENT REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED.

6. Approximation of Disclaimer and Damage Waiver IF THE DISCLAIMER AND DAMAGE WAIVER MENTIONED IN SECTION 4 AND SECTION 5 CANNOT BE GIVEN LEGAL EFFECT UNDER APPLICABLE LOCAL LAW, REVIEWING COURTS SHALL APPLY LOCAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE CONTRIBUTION.

7. Term 7.1 This Agreement shall come into effect upon Your acceptance of the terms and conditions.

7.2 In the event of a termination of this Agreement Sections 4, 5, 6, 7 and 8 shall survive such termination and shall remain in full force thereafter. For the avoidance of doubt, Contributions that are already licensed under a free and open source license at the date of the termination shall remain in full force after the termination of this Agreement.

8. Miscellaneous 8.1 This Agreement and all disputes, claims, actions, suits or other proceedings arising out of this agreement or relating in any way to it shall be governed by the laws of People's Republic of China excluding its private international law provisions.

8.2 This Agreement sets out the entire agreement between You and Us for Your Contributions to Us and overrides all other agreements or understandings.

8.3 If any provision of this Agreement is found void and unenforceable, such provision will be replaced to the extent possible with a provision that comes closest to the meaning of the original provision and that is enforceable. The terms and conditions set forth in this Agreement shall apply notwithstanding any failure of essential purpose of this Agreement or any limited remedy to the maximum extent possible under law.

8.4 You agree to notify Us of any facts or circumstances of which you become aware that would make this Agreement inaccurate in any respect.

You

Date:	
Name:	
Title:	
Address:	

Us

Date:	
Name:	
Title:	
Address:	

7.5.7 Copyright Header Guide

ESP-IDF is released under the Apache License 2.0 with some additional third-party copyrighted code released under various licenses. For further information please refer to *the list of copyrights and licenses*.

This page explains how the source code should be properly marked with a copyright header. ESP-IDF uses The Software Package Data Exchange (SPDX) format which is short and can be easily read by humans or processed by automated tools for copyright checks.

How to Check the Copyright Headers

Please make sure you have installed *the pre-commit hooks* which contain a copyright header checker as well. The checker can suggest a header if it is not able to detect a properly formatted SPDX header.

What if the Checker's Suggestion is Incorrect?

No automated checker (no matter how good is) can replace humans. So the developer's responsibility is to modify the offered header to be in line with the law and the license restrictions of the original code on which the work is based on. Certain licenses are not compatible between each other. Such corner cases will be covered by the following examples.

The checker can be configured with the `tools/ci/check_copyright_config.yaml` configuration file. Please check the options it offers and consider updating it in order to match the headers correctly.

Common Examples of Copyright Headers

The simplest case is when the code is not based on any licensed previous work, e.g. it was written completely from scratch. Such code can be decorated with the following copyright header and put under the license of ESP-IDF:

```
/*
 * SPDX-FileCopyrightText: 2015-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
```

Less restrictive parts of ESP-IDF Some parts of ESP-IDF are deliberately under less restrictive licenses in order to ease their re-use in commercial closed source projects. This is the case for ESP-IDF examples which are in Public domain or under the Creative Commons Zero Universal (CC0) license. The following header can be used in such source files:

```
/*
 * SPDX-FileCopyrightText: 2015-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Unlicense OR CC0-1.0
 */
```

The option allowing multiple licenses joined with the `OR` keyword from the above example can be achieved with the definition of multiple allowed licenses in the `tools/ci/check_copyright_config.yaml` configuration file. Please use this option with care and only selectively for a limited part of ESP-IDF.

Third party licenses Code licensed under different licenses, modified by Espressif Systems and included in ESP-IDF cannot be licensed under Apache License 2.0 not even if the checker suggests it. It is advised to keep the original copyright header and add an SPDX before it.

The following example is a suitable header for a code licensed under the "GNU General Public License v2.0 or later" held by John Doe with some additional modifications done by Espressif Systems:

```
/*
 * SPDX-FileCopyrightText: 1991 John Doe
 *
 * SPDX-License-Identifier: GPL-2.0-or-later
 *
 * SPDX-FileContributor: 2019-2022 Espressif Systems (Shanghai) CO LTD
 */
```

The licenses can be identified and the short SPDX identifiers can be found in the official SPDX license list. Other very common licenses are the GPL-2.0-only, the BSD-3-Clause, and the BSD-2-Clause.

The configuration stored in `tools/ci/check_copyright_config.yaml` offers features useful for third party licenses:

- A different license can be defined for the files part of a third party library.
- The check for a selected set of files can be permanently disabled. Please use this option with care and only in cases when none of the other options are suitable.

7.5.8 ESP-IDF Tests with Pytest Guide

This documentation is a guide that introduces the following aspects:

1. The basic idea of different test types in ESP-IDF

2. How to apply the pytest framework to the test python scripts to make sure the apps are working as expected.
3. ESP-IDF CI target test process
4. Run ESP-IDF tests with pytest locally
5. Tips and tricks on pytest

Disclaimer

In ESP-IDF, we use the following plugins by default:

- pytest-embedded with default services `esp,idf`
- pytest-rerunfailures

All the introduced concepts and usages are based on the default behavior in ESP-IDF. Not all of them are available in vanilla pytest.

Installation

All dependencies could be installed by running the install script with the `--enable-pytest` argument, e.g. `$ install.sh --enable-pytest`.

Common Issues During Installation

No Package 'dbus-1' found If you're facing an error message like:

```
configure: error: Package requirements (dbus-1 >= 1.8) were not met:

No package 'dbus-1' found

Consider adjusting the PKG_CONFIG_PATH environment variable if you
installed software in a non-standard prefix.
```

If you're running a ubuntu system, you may need to run:

```
sudo apt-get install libdbus-glib-1-dev
```

or

```
sudo apt-get install libdbus-1-dev
```

For other linux distros, you may Google the error message and find the solution. This issue could be solved by installing the related header files.

Invalid command 'bdist_wheel' If you're facing an error message like:

```
error: invalid command 'bdist_wheel'
```

You may need to run:

```
python -m pip install -U pip
```

Or

```
python -m pip install wheel
```

Before running the pip commands, please make sure you're using the IDF python virtual environment.

Basic Concepts

Component-based Unit Tests Component-based unit tests are our recommended way to test your component. All the test apps should be located under `${IDF_PATH}/components/<COMPONENT_NAME>/test_apps`.

For example:

```
components/
└── my_component/
    ├── include/
    │   └── ...
    ├── test_apps/
    │   ├── test_app_1
    │   │   ├── main/
    │   │   │   └── ...
    │   │   ├── CMakeLists.txt
    │   │   └── pytest_my_component_app_1.py
    │   ├── test_app_2
    │   │   ├── ...
    │   │   └── pytest_my_component_app_2.py
    │   └── parent_folder
    │       ├── test_app_3
    │       │   ├── ...
    │       │   └── pytest_my_component_app_3.py
    │       └── ...
    ├── my_component.c
    └── CMakeLists.txt
```

Example Tests Example Tests are tests for examples that are intended to demonstrate parts of the ESP-IDF functionality to our customers.

All the test apps should be located under `${IDF_PATH}/examples`. For more information please refer to the Examples Readme .

For example:

```
examples/
└── parent_folder/
    └── example_1/
        ├── main/
        │   └── ...
        ├── CMakeLists.txt
        └── pytest_example_1.py
```

Custom Tests Custom Tests are tests that aim to run some arbitrary test internally. They are not intended to demonstrate the ESP-IDF functionality to our customers in any way.

All the test apps should be located under `${IDF_PATH}/tools/test_apps`. For more information please refer to the Custom Test Readme .

Pytest in ESP-IDF

Pytest Execution Process

1. Bootstrapping Phase
   Create session-scoped caches:
   - port-target cache
   - port-app cache
2. Collection Phase
   1. Get all the python files with the prefix `pytest_`

2. Get all the test functions with the prefix `test_`
3. Apply the params, and duplicate the test functions.
4. Filter the test cases with CLI options. Introduced detailed usages *here*

3. Test Running Phase
   1. Construct the fixtures. In ESP-IDF, the common fixtures are initialized in this order:
      1. `pexpect_proc`: pexpect instance
      2. `app`: IdfApp instance
         The information of the app, like sdkconfig, flash_files, partition_table, etc., would be parsed at this phase.
      3. `serial`: IdfSerial instance
         The port of the host which connected to the target type parsed from the app would be auto-detected. The flash files would be auto flashed.
      4. `dut`: IdfDut instance
   2. Run the real test function
   3. Deconstruct the fixtures in this order:
      1. `dut`
         1. close the `serial` port
         2. (Only for apps with unity test framework) generate junit report of the unity test cases
      2. `serial`
      3. `app`
      4. `pexpect_proc`: Close the file descriptor
   4. (Only for apps with unity test framework)
      Raise `AssertionError` when detected unity test failed if you call `dut.expect_from_unity_output()` in the test function.
4. Reporting Phase
   1. Generate junit report of the test functions
   2. Modify the junit report test case name into ESP-IDF test case ID format: `<target>.<config>.<test function name>`
5. Finalizing Phase (Only for apps with unity test framework)
   Combine the junit reports if the junit reports of the unity test cases are generated.

Example Code This code example is taken from pytest_console_basic.py .

```
@pytest.mark.esp32
@pytest.mark.esp32c3
@pytest.mark.generic
@pytest.mark.parametrize('config', [
    'history',
    'nohistory',
], indirect=True)
def test_console_advanced(config: str, dut: Dut) -> None:
    if config == 'history':
        dut.expect('Command history enabled')
    elif config == 'nohistory':
        dut.expect('Command history disabled')
```

Note: Using `expect_exact` is better here. For further reading about the different types of `expect` functions, please refer to the pytest-embedded Expecting documentation.

Use Markers to Specify the Supported Targets You can use markers to specify the supported targets and the test env in CI. You can run `pytest --markers` to get more details about different markers.

```
@pytest.mark.esp32      # <-- support esp32
@pytest.mark.esp32c3    # <-- support esp32c3
@pytest.mark.generic    # <-- test env `generic, would assign to runner with tag
↪`generic`
```

Besides, if the test case supports all officially ESP-IDF-supported targets, like esp32, esp32s2, esp32s3, esp32c3 for now (2022.2), you can use a special marker `supported_targets` to apply them all in one line.

This code example is taken from pytest_gptimer_example.py .

```
@pytest.mark.supported_targets
@pytest.mark.generic
def test_gptimer_example(dut: Dut) -> None:
    ...
```

Use Params to Specify the sdkconfig Files You can use `pytest.mark.parametrize` with "config" to apply the same test to different apps with different sdkconfig files. For more information about `sdkconfig.ci.xxx` files, please refer to the Configuration Files section under this readme .

```
@pytest.mark.parametrize('config', [
    'history',      # <-- run with app built by sdkconfig.ci.history
    'nohistory',    # <-- run with app built by sdkconfig.ci.nohistory
], indirect=True)   # <-- `indirect=True` is required
```

Overall, this test function would be replicated to 4 test cases:

- esp32.history.test_console_advanced
- esp32.nohistory.test_console_advanced
- esp32c3.history.test_console_advanced
- esp32c3.nohistory.test_console_advanced

Advanced Examples

Multi Dut Tests with the Same App

```
@pytest.mark.esp32s2
@pytest.mark.esp32s3
@pytest.mark.usb_host
@pytest.mark.parametrize('count', [
    2,
], indirect=True)
def test_usb_host(dut: Tuple[IdfDut, IdfDut]) -> None:
    device = dut[0]  # <-- assume the first dut is the device
    host = dut[1]    # <-- and the second dut is the host
    ...
```

After setting the param `count` to 2, all these fixtures are changed into tuples.

Multi Dut Tests with Different Apps This code example is taken from pytest_wifi_getting_started.py .

```
@pytest.mark.esp32
@pytest.mark.multi_dut_generic
@pytest.mark.parametrize(
    'count, app_path', [
        (2,
         f'{os.path.join(os.path.dirname(__file__), "softAP")}|{os.path.join(os.
↪path.dirname(__file__), "station")}'),
    ], indirect=True
)
def test_wifi_getting_started(dut: Tuple[IdfDut, IdfDut]) -> None:
    softap = dut[0]
    station = dut[1]
    ...
```

Here the first dut was flashed with the app softap , and the second dut was flashed with the app station .

Note: Here the `app_path` should be set with absolute path. the `__file__` macro in python would return the absolute path of the test script itself.

Multi Dut Tests with Different Apps, and Targets This code example is taken from pytest_wifi_getting_started.py . As the comment says, for now it' s not running in the ESP-IDF CI.

```
@pytest.mark.parametrize(
    'count, app_path, target', [
        (2,
         f'{os.path.join(os.path.dirname(__file__), "softAP")}|{os.path.join(os.
↪path.dirname(__file__), "station")}',
         'esp32|esp32s2'),
        (2,
         f'{os.path.join(os.path.dirname(__file__), "softAP")}|{os.path.join(os.
↪path.dirname(__file__), "station")}',
         'esp32s2|esp32'),
    ],
    indirect=True,
)
def test_wifi_getting_started(dut: Tuple[IdfDut, IdfDut]) -> None:
    softap = dut[0]
    station = dut[1]
    ...
```

Overall, this test function would be replicated to 2 test cases:

- softap with esp32 target, and station with esp32s2 target
- softap with esp32s2 target, and station with esp32 target

Support different targets with different sdkconfig files This code example is taken from pytest_panic.py as an advanced example.

```
CONFIGS = [
    pytest.param('coredump_flash_bin_crc', marks=[pytest.mark.esp32, pytest.mark.
↪esp32s2]),
    pytest.param('coredump_flash_elf_sha', marks=[pytest.mark.esp32]),  # sha256␣
↪only supported on esp32
    pytest.param('coredump_uart_bin_crc', marks=[pytest.mark.esp32, pytest.mark.
↪esp32s2]),
    pytest.param('coredump_uart_elf_crc', marks=[pytest.mark.esp32, pytest.mark.
↪esp32s2]),
    pytest.param('gdbstub', marks=[pytest.mark.esp32, pytest.mark.esp32s2]),
    pytest.param('panic', marks=[pytest.mark.esp32, pytest.mark.esp32s2]),
]

@pytest.mark.parametrize('config', CONFIGS, indirect=True)
...
```

Use Custom Class Usually, you can write a custom class in these conditions:

1. Add more reusable functions for a certain number of DUTs
2. Add custom setup and teardown functions in different phases described *here*

This code example is taken from panic/conftest.py

```
class PanicTestDut(IdfDut):
    ...

@pytest.fixture(scope='module')
def monkeypatch_module(request: FixtureRequest) -> MonkeyPatch:
    mp = MonkeyPatch()
    request.addfinalizer(mp.undo)
    return mp


@pytest.fixture(scope='module', autouse=True)
def replace_dut_class(monkeypatch_module: MonkeyPatch) -> None:
    monkeypatch_module.setattr('pytest_embedded_idf.dut.IdfDut', PanicTestDut)
```

`monkeypatch_module` provide a module-scoped monkeypatch fixture.

`replace_dut_class` is a module-scoped autouse fixture. This function replaces the `IdfDut` class with your custom class.

Mark Flaky Tests Sometimes, our test is based on ethernet or wifi. The network may cause the test flaky. We could mark the single test case within the code repo.

This code example is taken from pytest_esp_eth.py

```
@pytest.mark.flaky(reruns=3, reruns_delay=5)
def test_esp_eth_ip101(dut: Dut) -> None:
    ...
```

This flaky marker means that if the test function failed, the test case would rerun for a maximum of 3 times with 5 seconds delay.

Mark Known Failure Cases Sometimes a test couldn't pass for the following reasons:

- Has a bug
- The success ratio is too low because of environment issue, such as network issue. Retry couldn't help

Now you may mark this test case with marker xfail with a user-friendly readable reason.

This code example is taken from pytest_panic.py

```
@pytest.mark.xfail('config.getvalue("target") == "esp32s2"', reason='raised␣
↪IllegalInstruction instead')
def test_cache_error(dut: PanicTestDut, config: str, test_func_name: str) -> None:
```

This marker means that if the test would be a known failure one on esp32s2.

Mark Nightly Run Test Cases Some tests cases are only triggered in nightly run pipelines due to a lack of runners.

```
@pytest.mark.nightly_run
```

This marker means that the test case would only be run with env var `NIGHTLY_RUN` or `INCLUDE_NIGHTLY_RUN`.

Run the Tests in CI

The workflow in CI is simple, build jobs -> target test jobs.

Build Jobs

Build Job Names

- Component-based Unit Tests: `build_pytest_components_<target>`
- Example Tests: `build_pytest_examples_<target>`
- Custom Tests: `build_pytest_test_apps_<target>`

Build Job Command The command used by CI to build all the relevant tests is: `python $IDF_PATH/tools/ci/ci_build_apps.py <parent_dir> --target <target> -vv --pytest-apps`

All apps which supported the specified target would be built with all supported sdkconfig files under `build_<target>_<config>`.

For example, If you run `python $IDF_PATH/tools/ci/ci_build_apps.py $IDF_PATH/examples/system/console/basic --target esp32 --pytest-apps`, the folder structure would be like this:

```
basic
├── build_esp32_history/
│   └── ...
├── build_esp32_nohistory/
│   └── ...
├── main/
├── CMakeLists.txt
├── pytest_console_basic.py
└── ...
```

All the binaries folders would be uploaded as artifacts under the same directories.

Target Test Jobs

Target Test Job Names

- Component-based Unit Tests: `component_ut_pytest_<target>_<test_env>`
- Example Tests: `example_test_pytest_<target>_<test_env>`
- Custom Tests: `test_app_test_pytest_<target>_<test_env>`

Target Test Job Command The command used by CI to run all the relevant tests is: `pytest <parent_dir> --target <target> -m <test_env_marker>`

All test cases with the specified target marker and the test env marker under the parent folder would be executed.

The binaries in the target test jobs are downloaded from build jobs, the artifacts would be placed under the same directories.

Run the Tests Locally

The local executing process is the same as the CI process.

For example, if you want to run all the esp32 tests under the `$IDF_PATH/examples/system/console/basic` folder, you may:

```
$ pip install pytest-embedded-serial-esp pytest-embedded-idf
$ cd $IDF_PATH
$ . ./export.sh
$ cd examples/system/console/basic
$ python $IDF_PATH/tools/ci/ci_build_apps.py . --target esp32 -vv --pytest-apps
$ pytest --target esp32
```

Tips and Tricks

Filter the Test Cases

- filter by target with `pytest --target <target>`
  pytest would run all the test cases that support specified target.
- filter by sdkconfig file with `pytest --sdkconfig <sdkconfig>`
  if `<sdkconfig>` is `default`, pytest would run all the test cases with the sdkconfig file `sdkconfig.defaults`.
  In other cases, pytest would run all the test cases with sdkconfig file `sdkconfig.ci.<sdkconfig>`.

Add New Markers We're using two types of custom markers, target markers which indicate that the test cases should support this target, and env markers which indicate that the test case should be assigned to runners with these tags in CI.

You can add new markers by adding one line under the `${IDF_PATH}/pytest.ini markers =` section. The grammar should be: `<marker_name>: <marker_description>`

Generate JUnit Report You can call pytest with `--junitxml <filepath>` to generate the JUnit report. In ESP-IDF, the test case name would be unified as "<target>.<config>.<function_name>".

Skip Auto Flash Binary Skipping auto-flash binary every time would be useful when you're debugging your test script.

You can call pytest with `--skip-autoflash y` to achieve it.

Record Statistics Sometimes you may need to record some statistics while running the tests, like the performance test statistics.

You can use record_xml_attribute fixture in your test script, and the statistics would be recorded as attributes in the JUnit report.

Logging System Sometimes you may need to add some extra logging lines while running the test cases.

You can use python logging module to achieve this.

Known Limitations and Workarounds

Avoid Using `Thread` for Performance Test `pytest-embedded` is using some threads internally to help gather all stdout to the pexpect process. Due to the limitation of Global Interpreter Lock, if you're using threads to do performance tests, these threads would block each other and there would be great performance loss.

workaround

Use Process instead, the APIs should be almost the same as `Thread`.

Further Readings

- pytest documentation: https://docs.pytest.org/en/latest/contents.html
- pytest-embedded documentation: https://docs.espressif.com/projects/pytest-embedded/en/latest/

Chapter 8

ESP-IDF Versions

The ESP-IDF GitHub repository is updated regularly, especially the master branch where new development takes place.

For production use, there are also stable releases available.

8.1 Releases

The documentation for the current stable release version can always be found at this URL:

https://docs.espressif.com/projects/esp-idf/en/stable/

Documentation for the latest version (master branch) can always be found at this URL:

https://docs.espressif.com/projects/esp-idf/en/latest/

The full history of releases can be found on the GitHub repository Releases page. There you can find release notes, links to each version of the documentation, and instructions for obtaining each version.

8.2 Which Version Should I Start With?

- For production purposes, use the current stable version. Stable versions have been manually tested, and are updated with "bugfix releases" which fix bugs without changing other functionality (see *Versioning Scheme* for more details). Every stable release version can be found on the Releases page.
- For prototyping, experimentation or for developing new ESP-IDF features, use the latest version (master branch in Git). The latest version in the master branch has all the latest features and has passed automated testing, but has not been completely manually tested ("bleeding edge").
- If a required feature is not yet available in a stable release, but you do not want to use the master branch, it is possible to check out a pre-release version or a release branch. It is recommended to start from a stable version and then follow the instructions for *Updating to a Pre-Release Version* or *Updating to a Release Branch*.
- If you plan to use another project which is based on ESP-IDF, please check the documentation of that project to determine the version(s) of ESP-IDF it is compatible with.

See *Updating ESP-IDF* if you already have a local copy of ESP-IDF and wish to update it.

8.3 Versioning Scheme

ESP-IDF uses Semantic Versioning. This means that:

- Major Releases, like `v3.0`, add new functionality and may change functionality. This includes removing deprecated functionality.
  If updating to a new major release (for example, from `v2.1` to `v3.0`), some of your project's code may need updating and functionality may need to be re-tested. The release notes on the Releases page include lists of Breaking Changes to refer to.
- Minor Releases like `v3.1` add new functionality and fix bugs but will not change or remove documented functionality, or make incompatible changes to public APIs.
  If updating to a new minor release (for example, from `v3.0` to `v3.1`), your project's code does not require updating, but you should re-test your project. Pay particular attention to the items mentioned in the release notes on the Releases page.
- Bugfix Releases like `v3.0.1` only fix bugs and do not add new functionality.
  If updating to a new bugfix release (for example, from `v3.0` to `v3.0.1`), you do not need to change any code in your project, and you only need to re-test the functionality directly related to bugs listed in the release notes on the Releases page.

8.4 Support Periods

Each ESP-IDF major and minor release version has an associated support period. After this period, the release is End of Life and no longer supported.

The ESP-IDF Support Period Policy explains this in detail, and describes how the support periods for each release are determined.

Each release on the Releases page includes information about the support period for that particular release.

As a general guideline:

- If starting a new project, use the latest stable release.
- If you have a GitHub account, click the "Watch" button in the top-right of the Releases page and choose "Releases only". GitHub will notify you whenever a new release is available. Whenever a bug fix release is available for the version you are using, plan to update to it.
- If possible, periodically update the project to a new major or minor ESP-IDF version (for example, once a year.) The update process should be straightforward for Minor updates, but may require some planning and checking of the release notes for Major updates.
- Always plan to update to a newer release before the release you are using becomes End of Life.

Each ESP-IDF major and minor release (V4.1, V4.2, etc) is supported for 30 months after the initial stable release date.

Supported means that the ESP-IDF team will continue to apply bug fixes, security fixes, etc to the release branch on GitHub, and periodically make new bugfix releases as needed.

Support period is divided into "Service" and "Maintenance" period:

Period	Duration	Recommended for new projects?
Service	12 months	Yes
Maintenance	18 months	No

During the Service period, bugfixes releases are more frequent. In some cases, support for new features may be added during the Service period (this is reserved for features which are needed to meet particular regulatory requirements or standards for new products, and which carry a very low risk of introducing regressions.)

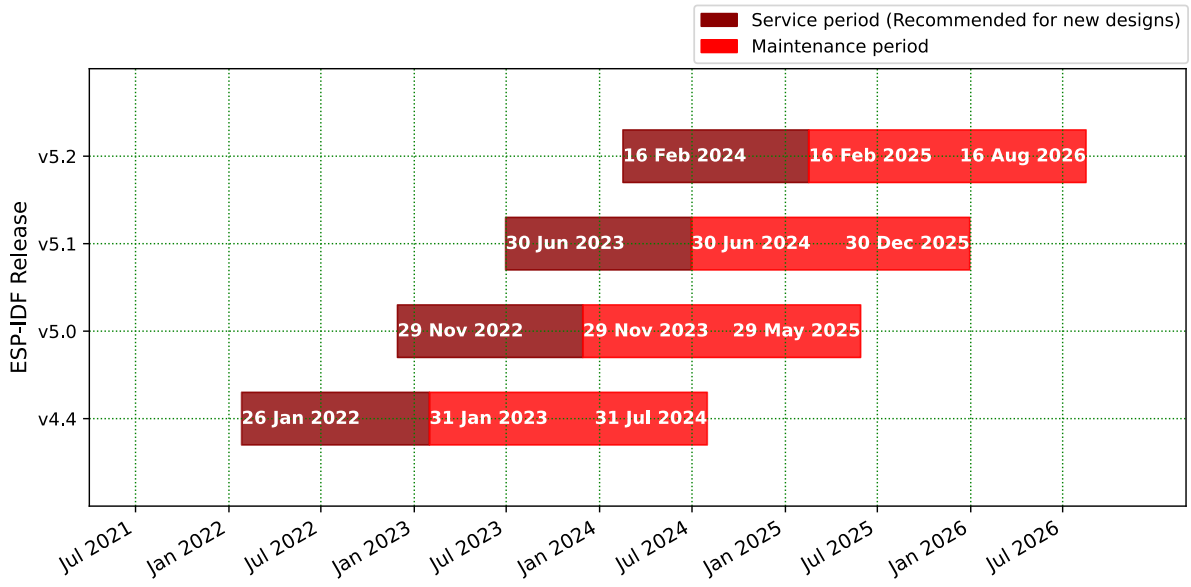
During the Maintenance period, the version is still supported but only bugfixes for high severity issues or security issues will be applied.

Using an "In Service" version is recommended when starting a new project.

Users are encouraged to upgrade all projects to a newer ESP-IDF release before the support period finishes and the release becomes End of Life (EOL). It is our policy to not continue fixing bugs in End of Life releases.

Pre-release versions (betas, previews, *-rc* and *-dev* versions, etc) are not covered by any support period. Sometimes a particular feature is marked as "Preview" in a release, which means it is also not covered by the support period.

The ESP-IDF Programming Guide has information about the different versions of ESP-IDF (major, minor, bugfix, etc).



8.5 Checking the Current Version

The local ESP-IDF version can be checked by using idf.py:

```
idf.py --version
```

The ESP-IDF version is also compiled into the firmware and can be accessed (as a string) via the macro `IDF_VER`. The default ESP-IDF bootloader will print the version on boot (the version information is not always updated if the code in the GitHub repo is updated, it only changes if there is a clean build or if that particular source file is recompiled).

If writing code that needs to support multiple ESP-IDF versions, the version can be checked at compile time using *compile-time macros*.

Examples of ESP-IDF versions:

Version String	Meaning
`v3.2-dev-306-gbeb3611ca`	Master branch pre-release. - `v3.2-dev` - in development for version 3.2. - `306` - number of commits after v3.2 development started. - `beb3611ca` - commit identifier.
`v3.0.2`	Stable release, tagged `v3.0.2`.
`v3.1-beta1-75-g346d6b0ea`	Beta version in development (on a *release branch*). - `v3.1-beta1` - pre-release tag. - `75` - number of commits after the pre-release beta tag was assigned. - `346d6b0ea` - commit identifier.
`v3.0.1-dirty`	Stable release, tagged `v3.0.1`. - `dirty` means that there are modifications in the local ESP-IDF directory.

8.6 Git Workflow

The development (Git) workflow of the Espressif ESP-IDF team is as follows:

- New work is always added on the master branch (latest version) first. The ESP-IDF version on `master` is always tagged with `-dev` (for "in development"), for example `v3.1-dev`.
- Changes are first added to an internal Git repository for code review and testing but are pushed to GitHub after automated testing passes.
- When a new version (developed on `master`) becomes feature complete and "beta" quality, a new branch is made for the release, for example `release/v3.1`. A pre-release tag is also created, for example `v3.1-beta1`. You can see a full list of branches and a list of tags on GitHub. Beta pre-releases have release notes which may include a significant number of Known Issues.
- As testing of the beta version progresses, bug fixes will be added to both the `master` branch and the release branch. New features for the next release may start being added to `master` at the same time.
- Once testing is nearly complete a new release candidate is tagged on the release branch, for example `v3.1-rc1`. This is still a pre-release version.
- If no more significant bugs are found or reported, then the final Major or Minor Version is tagged, for example `v3.1`. This version appears on the Releases page.
- As bugs are reported in released versions, the fixes will continue to be committed to the same release branch.
- Regular bugfix releases are made from the same release branch. After manual testing is complete, a bugfix release is tagged (i.e. `v3.1.1`) and appears on the Releases page.

8.7 Updating ESP-IDF

Updating ESP-IDF depends on which version(s) you wish to follow:

- *Updating to Stable Release* is recommended for production use.

- *Updating to Master Branch* is recommended for the latest features, development use, and testing.
- *Updating to a Release Branch* is a compromise between the first two.

Note: These guides assume that you already have a local copy of ESP-IDF cloned. To get one, check Step 2 in the *Getting Started* guide for any ESP-IDF version.

8.7.1 Updating to Stable Release

To update to a new ESP-IDF release (recommended for production use), this is the process to follow:

- Check the Releases page regularly for new releases.
- When a bugfix release for the version you are using is released (for example, if using `v3.0.1` and `v3.0.2` is released), check out the new bugfix version into the existing ESP-IDF directory:

```
cd $IDF_PATH
git fetch
git checkout vX.Y.Z
git submodule update --init --recursive
```

- When major or minor updates are released, check the Release Notes on the releases page and decide if you want to update or to stay with your current release. Updating is via the same Git commands shown above.

Note: If you installed the stable release via zip file instead of using git, it might not be possible to update versions using the commands. In this case, update by downloading a new zip file and replacing the entire `IDF_PATH` directory with its contents.

8.7.2 Updating to a Pre-Release Version

It is also possible to `git checkout` a tag corresponding to a pre-release version or release candidate, the process is the same as *Updating to Stable Release*.

Pre-release tags are not always found on the Releases page. Consult the list of tags on GitHub for a full list. Caveats for using a pre-release are similar to *Updating to a Release Branch*.

8.7.3 Updating to Master Branch

Note: Using Master branch means living "on the bleeding edge" with the latest ESP-IDF code.

To use the latest version on the ESP-IDF master branch, this is the process to follow:

- Check out the master branch locally:

```
cd $IDF_PATH
git checkout master
git pull
git submodule update --init --recursive
```

- Periodically, re-run `git pull` to pull the latest version of master. Note that you may need to change your project or report bugs after updating your master branch.
- To switch from master to a release branch or stable version, run `git checkout` as shown in the other sections.

Important: It is strongly recommended to regularly run `git pull` and then `git submodule update --init --recursive` so a local copy of master does not get too old. Arbitrary old master branch revisions are effectively unsupportable "snapshots" that may have undocumented bugs. For a semi-stable version, try *Updating to a Release Branch* instead.

8.7.4 Updating to a Release Branch

In terms of stability, using a release branch is part-way between using the master branch and only using stable releases. A release branch is always beta quality or better, and receives bug fixes before they appear in each stable release.

You can find a list of branches on GitHub.

For example, to follow the branch for ESP-IDF v3.1, including any bugfixes for future releases like `v3.1.1`, etc:

```
cd $IDF_PATH
git fetch
git checkout release/v3.1
git pull
git submodule update --init --recursive
```

Each time you `git pull` this branch, ESP-IDF will be updated with fixes for this release.

Note: There is no dedicated documentation for release branches. It is recommended to use the documentation for the closest version to the branch which is currently checked out.

Chapter 9

Resources

9.1 PlatformIO

- *What is PlatformIO?*
- *Installation*
- *Configuration*
- *Tutorials*
- *Project Examples*
- *Next Steps*

9.1.1 What is PlatformIO?

PlatformIO is a cross-platform embedded development environment with out-of-the-box support for ESP-IDF.

Since ESP-IDF support within PlatformIO is not maintained by the Espressif team, please report any issues with PlatformIO directly to its developers in the official PlatformIO repositories.

A detailed overview of the PlatformIO ecosystem and its philosophy can be found in the official PlatformIO documentation.

9.1.2 Installation

- PlatformIO IDE is a toolset for embedded C/C++ development available on Windows, macOS and Linux platforms
- PlatformIO Core (CLI) is a command-line tool that consists of multi-platform build system, platform and library managers and other integration components. It can be used with a variety of code development environments and allows integration with cloud platforms and web services

9.1.3 Configuration

Please go through the official PlatformIO configuration guide for ESP-IDF.

9.1.4 Tutorials

- ESP-IDF and ESP32-DevKitC: debugging, unit testing, project analysis

9.1.5 Project Examples

Please check ESP-IDF page in the official PlatformIO documentation

9.1.6 Next Steps

Here are some useful links for exploring the PlatformIO ecosystem:

- Learn more about integrations with other IDEs/Text Editors
- Get help from PlatformIO community

9.2 Useful Links

- The esp32.com forum is a place to ask questions and find community resources.
- Check the Issues section on GitHub if you find a bug or have a feature request. Please check existing Issues before opening a new one.
- A comprehensive collection of solutions, practical applications, components and drivers based on ESP-IDF is available in ESP IoT Solution repository. In most of cases descriptions are provided both in English and in 中文.
- To develop applications using Arduino platform, refer to Arduino core for the ESP32, ESP32-S2 and ESP32-C3.
- Several books have been written about ESP32 and they are listed on Espressif web site.
- If you're interested in contributing to ESP-IDF, please check the *Contributions Guide*.
- For additional ESP32 product related information, please refer to documentation section of Espressif site.
- Download latest and previous versions of this documentation in PDF and HTML format.

Chapter 10

Copyrights and Licenses

10.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2022 Espressif Systems. This source code is licensed under the Apache License 2.0 as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

Some examples use external components which are not Apache licensed, please check the copyright description in each example source code.

10.1.1 Firmware Components

These third party libraries can be included into the application (firmware) produced by ESP-IDF.

- Newlib is licensed under the BSD License and is Copyright of various parties, as described in COPYING.NEWLIB.
- Xtensa header files are Copyright (C) 2013 Tensilica Inc and are licensed under the MIT License as reproduced in the individual header files.
- Original parts of FreeRTOS (components/freertos) are Copyright (C) 2017 Amazon.com, Inc. or its affiliates are licensed under the MIT License, as described in license.txt.
- Original parts of LWIP (components/lwip) are Copyright (C) 2001, 2002 Swedish Institute of Computer Science and are licensed under the BSD License as described in COPYING file.
- wpa_supplicant Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- Fast PBKDF2 Copyright (c) 2015 Joseph Birr-Pixton and licensed under CC0 Public Domain Dedication license.
- FreeBSD net80211 Copyright (c) 2004-2008 Sam Leffler, Errno Consulting and licensed under the BSD license.
- argtable3 argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license. argtable3 also includes the following software components. For details, please see argtable3 LICENSE file.
  - C Hash Table library, Copyright (c) 2002, Christopher Clark and licensed under 3-clause BSD license.
  - The Better String library, Copyright (c) 2014, Paul Hsieh and licensed under 3-clause BSD license.
  - TCL library, Copyright the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, ActiveState Corporation and other parties, and licensed under TCL/TK License.
- linenoise line editing library Copyright (c) 2010-2014 Salvatore Sanfilippo, Copyright (c) 2010-2013 Pieter Noordhuis, licensed under 2-clause BSD license.
- FatFS library, Copyright (C) 2017 ChaN, is licensed under a BSD-style license .
- cJSON library, Copyright (c) 2009-2017 Dave Gamble and cJSON contributors, is licensed under MIT license as described in LICENSE file .

- micro-ecc library, Copyright (c) 2014 Kenneth MacKay, is licensed under 2-clause BSD license.
- Mbed TLS library, Copyright (C) 2006-2018 ARM Limited, is licensed under Apache License 2.0 as described in LICENSE file .
- SPIFFS library, Copyright (c) 2013-2017 Peter Andersson, is licensed under MIT license as described in LICENSE file .
- SD/MMC driver is derived from OpenBSD SD/MMC driver, Copyright (c) 2006 Uwe Stuehler, and is licensed under BSD license.
- ESP-MQTT MQTT Package (contiki-mqtt) - Copyright (c) 2014, Stephen Robinson, MQTT-ESP - Tuan PM <tuanpm at live dot com> is licensed under Apache License 2.0 as described in LICENSE file .
- BLE Mesh is adapted from Zephyr Project, Copyright (c) 2017-2018 Intel Corporation and licensed under Apache License 2.0.
- mynewt-nimble Apache Mynewt NimBLE, Copyright 2015-2018, The Apache Software Foundation, is licensed under Apache License 2.0 as described in LICENSE file.
- TLSF allocator Two Level Segregated Fit memory allocator, Copyright (c) 2006-2016, Matthew Conte, and licensed under the BSD 3-clause license.
- openthread, Copyright (c) The OpenThread Authors, is licensed under BSD License as described in LICENSE file.
- UBSAN runtime —Copyright (c) 2016, Linaro Limited and Jiří Zárevúcky, licensed under the BSD 2-clause license.
- HTTP Parser Based on src/http/ngx_http_parse.c from NGINX copyright Igor Sysoev. Additional changes are licensed under the same terms as NGINX and Joyent, Inc. and other Node contributors. For details please check LICENSE file.
- SEGGER SystemView target-side library, Copyright (c) 2015-2017 SEGGER Microcontroller GmbH & Co. KG, is licensed under BSD 3-clause license.

10.1.2 Documentation

- HTML version of the ESP-IDF Programming Guide uses the Sphinx theme sphinx_idf_theme, which is Copyright (c) 2013-2020 Dave Snider, Read the Docs, Inc. & contributors, and Espressif Systems (Shanghai) CO., LTD. It is based on sphinx_rtd_theme. Both are licensed under MIT license.

10.2 ROM Source Code Copyrights

ESP32, ESP32-S and ESP32-C Series SoCs mask ROM hardware includes binaries compiled from portions of the following third party software:

- Newlib , licensed under the BSD License and is Copyright of various parties, as described in COPYING.NEWLIB.
- Xtensa libhal, Copyright (c) Tensilica Inc and licensed under the MIT license (see below).
- TinyBasic Plus, Copyright Mike Field & Scott Lawrence and licensed under the MIT license (see below).
- miniz, by Rich Geldreich - placed into the public domain.
- wpa_supplicant Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- TJpgDec Copyright (C) 2011, ChaN, all right reserved. See below for license.
- **Parts of Zephyr RTOS USB stack:**
  - DesignWare USB device driver Copyright (c) 2016 Intel Corporation and licensed under Apache 2.0 license.
  - Generic USB device driver Copyright (c) 2006 Bertrik Sikken (bertrik@sikken.nl), 2016 Intel Corporation and licensed under BSD 3-clause license.
  - USB descriptors functionality Copyright (c) 2017 PHYTEC Messtechnik GmbH, 2017-2018 Intel Corporation and licensed under Apache 2.0 license.
  - USB DFU class driver Copyright(c) 2015-2016 Intel Corporation, 2017 PHYTEC Messtechnik GmbH and licensed under BSD 3-clause license.
  - USB CDC ACM class driver Copyright(c) 2015-2016 Intel Corporation and licensed under Apache 2.0 license

10.3 Xtensa libhal MIT License

Copyright (c) 2003, 2006, 2010 Tensilica Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10.4 TinyBasic Plus MIT License

Copyright (c) 2012-2013

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10.5 TJpgDec License

TJpgDec - Tiny JPEG Decompressor R0.01 (C)ChaN, 2011 The TJpgDec is a generic JPEG decompressor module for tiny embedded systems. This is a free software that opened for education, research and commercial developments under license policy of following terms.

Copyright (C) 2011, ChaN, all right reserved.

- The TJpgDec module is a free software and there is NO WARRANTY.
- No restriction on use. You can use, modify and redistribute it for personal, non-profit or commercial products UNDER YOUR RESPONSIBILITY.
- Redistributions of source code must retain the above copyright notice.

Chapter 11

About

This is documentation of ESP-IDF, the framework to develop applications for ESP32 chip by Espressif.

The ESP32 is 2.4 GHz Wi-Fi and Bluetooth combo, which integrates one or two Xtensa® 32-bit LX6 CPU, with up to 600 DMIPS processing power.

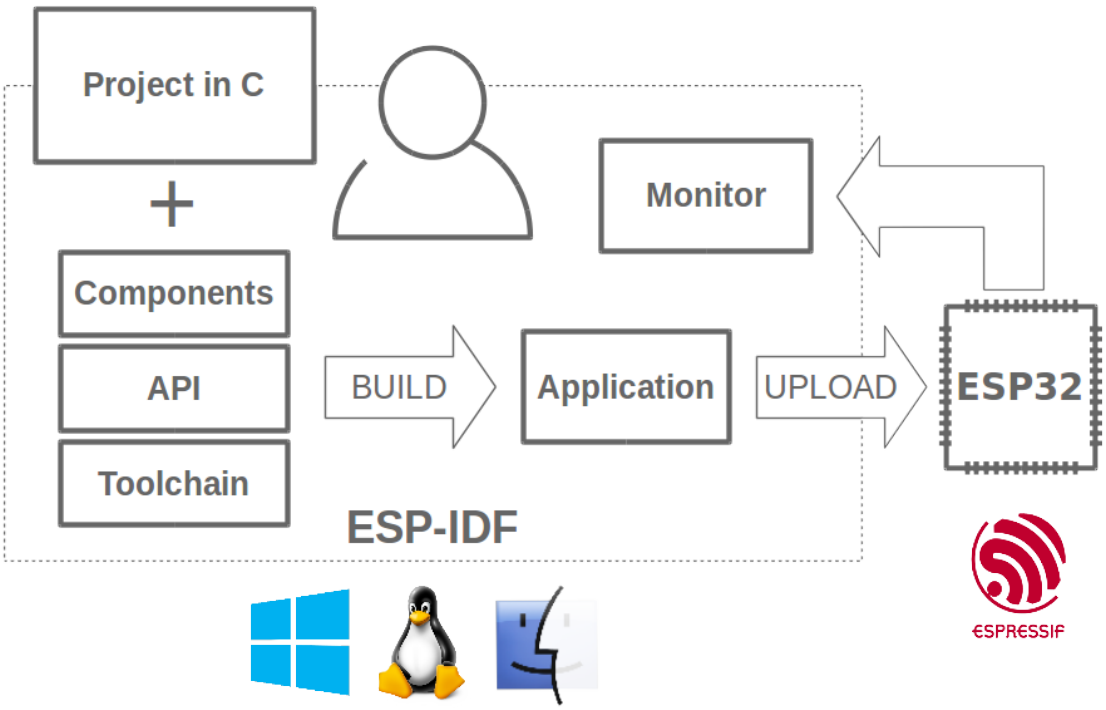


Fig. 1: Espressif IoT Integrated Development Framework

The ESP-IDF, Espressif IoT Development Framework, provides toolchain, API, components and workflows to develop applications for ESP32 using Windows, Linux and macOS operating systems.

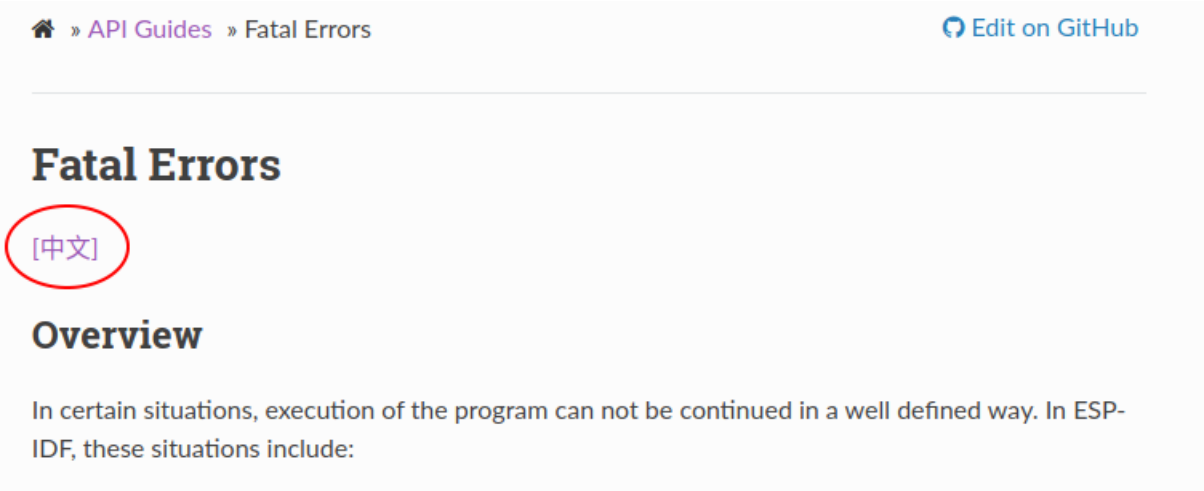
Chapter 12

Switch Between Languages

The ESP-IDF Programming Guide is now available in two languages. Please refer to the English version if there is any discrepancy.

- English
- Chinese

You can easily change from one language to another by clicking the language link you can find at the top of every document that has a translation.



Index

Symbols

_ESP_LOG_EARLY_ENABLED (*C macro*), 2079
_ip_addr (*C++ struct*), 1014
_ip_addr::ip4 (*C++ member*), 1014
_ip_addr::ip6 (*C++ member*), 1014
_ip_addr::type (*C++ member*), 1014
_ip_addr::u_addr (*C++ member*), 1014
[anonymous] (*C++ enum*), 492, 528, 606, 1792, 2134
[anonymous]::ESP_BLE_MESH_SERVER_FLAG_MAX (*C++ enumerator*), 606
[anonymous]::ESP_BLE_MESH_SERVER_TRANS_TIMER_START (*C++ enumerator*), 606
[anonymous]::ESP_BLE_SCA_100PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_150PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_20PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_250PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_30PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_500PPM (*C++ enumerator*), 528
[anonymous]::ESP_BLE_SCA_50PPM (*C++ enumerator*), 529
[anonymous]::ESP_BLE_SCA_75PPM (*C++ enumerator*), 529
[anonymous]::ESP_ERR_FLASH_NO_RESPONSE (*C++ enumerator*), 1792
[anonymous]::ESP_ERR_FLASH_SIZE_NOT_MATCH (*C++ enumerator*), 1792
[anonymous]::ESP_ERR_SLEEP_REJECT (*C++ enumerator*), 2134
[anonymous]::ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION (*C++ enumerator*), 2134
[anonymous]::ESP_HIDD_BOOT_REPORT_SIZE_KEYBOARD (*C++ enumerator*), 492
[anonymous]::ESP_HIDD_BOOT_REPORT_SIZE_MOUSE (*C++ enumerator*), 492

A

adc_atten_t (*C++ enum*), 1026
adc_atten_t::ADC_ATTEN_DB_0 (*C++ enumerator*), 1026
adc_atten_t::ADC_ATTEN_DB_11 (*C++ enumerator*), 1027
adc_atten_t::ADC_ATTEN_DB_12 (*C++ enumerator*), 1027
adc_atten_t::ADC_ATTEN_DB_2_5 (*C++ enumerator*), 1027
adc_atten_t::ADC_ATTEN_DB_6 (*C++ enumerator*), 1027
adc_bitwidth_t (*C++ enum*), 1027
adc_bitwidth_t::ADC_BITWIDTH_10 (*C++ enumerator*), 1027
adc_bitwidth_t::ADC_BITWIDTH_11 (*C++ enumerator*), 1027
adc_bitwidth_t::ADC_BITWIDTH_12 (*C++ enumerator*), 1027
adc_bitwidth_t::ADC_BITWIDTH_13 (*C++ enumerator*), 1027
adc_bitwidth_t::ADC_BITWIDTH_9 (*C++ enumerator*), 1027
adc_bitwidth_t::ADC_BITWIDTH_DEFAULT (*C++ enumerator*), 1027
adc_cali_check_scheme (*C++ function*), 1040
adc_cali_handle_t (*C++ type*), 1041
adc_cali_raw_to_voltage (*C++ function*), 1040
adc_cali_scheme_ver_t (*C++ enum*), 1041
adc_cali_scheme_ver_t::ADC_CALI_SCHEME_VER_CURVE_ (*C++ enumerator*), 1041
adc_cali_scheme_ver_t::ADC_CALI_SCHEME_VER_LINE_F (*C++ enumerator*), 1041
adc_channel_t (*C++ enum*), 1026
adc_channel_t::ADC_CHANNEL_0 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_1 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_2 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_3 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_4 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_5 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_6 (*C++ enumerator*), 1026
adc_channel_t::ADC_CHANNEL_7 (*C++ enumerator*), 1026

adc_channel_t::ADC_CHANNEL_8 (C++ enumerator), 1026
adc_channel_t::ADC_CHANNEL_9 (C++ enumerator), 1026
adc_continuous_callback_t (C++ type), 1038
adc_continuous_channel_to_io (C++ function), 1036
adc_continuous_config (C++ function), 1035
adc_continuous_config_t (C++ struct), 1037
adc_continuous_config_t::adc_pattern (C++ member), 1037
adc_continuous_config_t::conv_mode (C++ member), 1037
adc_continuous_config_t::format (C++ member), 1037
adc_continuous_config_t::pattern_num (C++ member), 1037
adc_continuous_config_t::sample_freq_hz (C++ member), 1037
adc_continuous_deinit (C++ function), 1036
adc_continuous_evt_cbs_t (C++ struct), 1037
adc_continuous_evt_cbs_t::on_conv_done (C++ member), 1038
adc_continuous_evt_cbs_t::on_pool_ovf (C++ member), 1038
adc_continuous_evt_data_t (C++ struct), 1037
adc_continuous_evt_data_t::conv_frame_buffer (C++ member), 1037
adc_continuous_evt_data_t::size (C++ member), 1037
adc_continuous_handle_cfg_t (C++ struct), 1036
adc_continuous_handle_cfg_t::conv_frame_size (C++ member), 1037
adc_continuous_handle_cfg_t::max_store_buf_size (C++ member), 1036
adc_continuous_handle_t (C++ type), 1038
adc_continuous_io_to_channel (C++ function), 1036
adc_continuous_new_handle (C++ function), 1034
adc_continuous_read (C++ function), 1035
adc_continuous_register_event_callbacks (C++ function), 1035
adc_continuous_start (C++ function), 1035
adc_continuous_stop (C++ function), 1036
adc_digi_convert_mode_t (C++ enum), 1027
adc_digi_convert_mode_t::ADC_CONV_ALTER_UNIT (C++ enumerator), 1028
adc_digi_convert_mode_t::ADC_CONV_BOTH_UNIT (C++ enumerator), 1028
adc_digi_convert_mode_t::ADC_CONV_SINGLE_UNIT_1 (C++ enumerator), 1028
adc_digi_convert_mode_t::ADC_CONV_SINGLE_UNIT_2 (C++ enumerator), 1028
adc_digi_output_data_t (C++ struct), 1025
adc_digi_output_data_t::channel (C++ member), 1025
adc_digi_output_data_t::data (C++ member), 1025
adc_digi_output_data_t::type1 (C++ member), 1025
adc_digi_output_data_t::type2 (C++ member), 1025
adc_digi_output_data_t::unit (C++ member), 1025
adc_digi_output_data_t::val (C++ member), 1025
adc_digi_output_format_t (C++ enum), 1028
adc_digi_output_format_t::ADC_DIGI_OUTPUT_FORMAT_ (C++ enumerator), 1028
adc_digi_output_format_t::ADC_DIGI_OUTPUT_FORMAT_ (C++ enumerator), 1028
adc_digi_pattern_config_t (C++ struct), 1024
adc_digi_pattern_config_t::atten (C++ member), 1025
adc_digi_pattern_config_t::bit_width (C++ member), 1025
adc_digi_pattern_config_t::channel (C++ member), 1025
adc_digi_pattern_config_t::unit (C++ member), 1025
ADC_MAX_DELAY (C macro), 1038
adc_oneshot_chan_cfg_t (C++ struct), 1030
adc_oneshot_chan_cfg_t::atten (C++ member), 1030
adc_oneshot_chan_cfg_t::bitwidth (C++ member), 1030
adc_oneshot_channel_to_io (C++ function), 1029
adc_oneshot_config_channel (C++ function), 1028
adc_oneshot_del_unit (C++ function), 1029
adc_oneshot_io_to_channel (C++ function), 1029
adc_oneshot_new_unit (C++ function), 1028
adc_oneshot_read (C++ function), 1029
adc_oneshot_unit_handle_t (C++ type), 1030
adc_oneshot_unit_init_cfg_t (C++ struct), 1030
adc_oneshot_unit_init_cfg_t::ulp_mode (C++ member), 1030
adc_oneshot_unit_init_cfg_t::unit_id (C++ member), 1030
adc_ulp_mode_t (C++ enum), 1027
adc_ulp_mode_t::ADC_ULP_MODE_DISABLE (C++ enumerator), 1027
adc_ulp_mode_t::ADC_ULP_MODE_FSM (C++ enumerator), 1027
adc_ulp_mode_t::ADC_ULP_MODE_RISCV (C++ enumerator), 1027
adc_unit_t (C++ enum), 1026
adc_unit_t::ADC_UNIT_1 (C++ enumerator), 1026

adc_unit_t::ADC_UNIT_2 (*C++ enumerator*), 1026
ADD_DEV_FLUSHABLE_DEV_FLAG (*C macro*), 579
ADD_DEV_RM_AFTER_PROV_FLAG (*C macro*), 578
ADD_DEV_START_PROV_NOW_FLAG (*C macro*), 578
ALU_SEL_ADD (*C macro*), 2172
ALU_SEL_AND (*C macro*), 2172
ALU_SEL_LSH (*C macro*), 2172
ALU_SEL_MOV (*C macro*), 2172
ALU_SEL_OR (*C macro*), 2172
ALU_SEL_RSH (*C macro*), 2172
ALU_SEL_SDEC (*C macro*), 2172
ALU_SEL_SINC (*C macro*), 2172
ALU_SEL_SRST (*C macro*), 2172
ALU_SEL_SUB (*C macro*), 2172

B

B_CMP_GE (*C macro*), 2173
B_CMP_L (*C macro*), 2173
BD_ADDR (*C++ type*), 597
BD_ADDR_LEN (*C macro*), 578
BLE_BIT (*C macro*), 300
BLE_DTM_PKT_PAYLOAD_0x00 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x01 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x02 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x03 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x04 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x05 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x06 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_0x07 (*C macro*), 299
BLE_DTM_PKT_PAYLOAD_MAX (*C macro*), 299
BLE_HCI_UART_H4_ACL (*C macro*), 816
BLE_HCI_UART_H4_CMD (*C macro*), 816
BLE_HCI_UART_H4_EVT (*C macro*), 816
BLE_HCI_UART_H4_NONE (*C macro*), 816
BLE_HCI_UART_H4_SCO (*C macro*), 816
BLE_UUID128_VAL_LENGTH (*C macro*), 1689
bluetooth_sdp_hdr_overlay (*C++ struct*), 518
bluetooth_sdp_hdr_overlay::l2cap_psm (*C++ member*), 518
bluetooth_sdp_hdr_overlay::profile_version (*C++ member*), 518
bluetooth_sdp_hdr_overlay::rfcomm_channel_number (*C++ member*), 518
bluetooth_sdp_hdr_overlay::service_name (*C++ member*), 518
bluetooth_sdp_hdr_overlay::service_name_length (*C++ member*), 518
bluetooth_sdp_hdr_overlay::type (*C++ member*), 518
bluetooth_sdp_hdr_overlay::user1_ptr (*C++ member*), 518
bluetooth_sdp_hdr_overlay::user1_ptr_len (*C++ member*), 518
bluetooth_sdp_hdr_overlay::user2_ptr (*C++ member*), 518
bluetooth_sdp_hdr_overlay::user2_ptr_len (*C++ member*), 518
bluetooth_sdp_hdr_overlay::uuid (*C++ member*), 518
bluetooth_sdp_mas_record (*C++ struct*), 519
bluetooth_sdp_mas_record::hdr (*C++ member*), 519
bluetooth_sdp_mas_record::mas_instance_id (*C++ member*), 519
bluetooth_sdp_mas_record::supported_features (*C++ member*), 519
bluetooth_sdp_mas_record::supported_message_types (*C++ member*), 519
bluetooth_sdp_mns_record (*C++ struct*), 519
bluetooth_sdp_mns_record::hdr (*C++ member*), 519
bluetooth_sdp_mns_record::supported_features (*C++ member*), 519
bluetooth_sdp_ops_record (*C++ struct*), 520
bluetooth_sdp_ops_record::hdr (*C++ member*), 520
bluetooth_sdp_ops_record::supported_formats_list (*C++ member*), 520
bluetooth_sdp_ops_record::supported_formats_list_ (*C++ member*), 520
bluetooth_sdp_pce_record (*C++ struct*), 519
bluetooth_sdp_pce_record::hdr (*C++ member*), 520
bluetooth_sdp_pse_record (*C++ struct*), 519
bluetooth_sdp_pse_record::hdr (*C++ member*), 519
bluetooth_sdp_pse_record::supported_features (*C++ member*), 519
bluetooth_sdp_pse_record::supported_repositories (*C++ member*), 519
bluetooth_sdp_sap_record (*C++ struct*), 520
bluetooth_sdp_sap_record::hdr (*C++ member*), 520
bootloader_fill_random (*C++ function*), 2122
bootloader_random_disable (*C++ function*), 2122
bootloader_random_enable (*C++ function*), 2121
bridgeif_config (*C++ struct*), 1007
bridgeif_config::max_fdb_dyn_entries (*C++ member*), 1007
bridgeif_config::max_fdb_sta_entries (*C++ member*), 1007
bridgeif_config::max_ports (*C++ member*), 1007
bridgeif_config_t (*C++ type*), 1010
BT_CONTROLLER_INIT_CONFIG_DEFAULT (*C macro*), 528
BT_OCTET32 (*C++ type*), 597
BT_OCTET32_LEN (*C macro*), 578
BT_SPP_DEFAULT_CONFIG (*C macro*), 440
BTHH_MAX_DSC_LEN (*C macro*), 504
btm_query_reason (*C++ enum*), 939

btm_query_reason::REASON_BANDWIDTH (C++ enumerator), 939
btm_query_reason::REASON_DELAY (C++ enumerator), 939
btm_query_reason::REASON_FRAME_LOSS (C++ enumerator), 939
btm_query_reason::REASON_GRAY_ZONE (C++ enumerator), 939
btm_query_reason::REASON_INTERFERENCE (C++ enumerator), 939
btm_query_reason::REASON_LOAD_BALANCE (C++ enumerator), 939
btm_query_reason::REASON_PREMIUM_AP (C++ enumerator), 939
btm_query_reason::REASON_RETRANSMISSION (C++ enumerator), 939
btm_query_reason::REASON_RSSI (C++ enumerator), 939
btm_query_reason::REASON_UNSPECIFIED (C++ enumerator), 939
BX_JUMP_TYPE_DIRECT (C macro), 2173
BX_JUMP_TYPE_OVF (C macro), 2173
BX_JUMP_TYPE_ZERO (C macro), 2173

C

CHIP_FEATURE_BLE (C macro), 2089
CHIP_FEATURE_BT (C macro), 2089
CHIP_FEATURE_EMB_FLASH (C macro), 2089
CHIP_FEATURE_EMB_PSRAM (C macro), 2089
CHIP_FEATURE_IEEE802154 (C macro), 2089
CHIP_FEATURE_WIFI_BGN (C macro), 2089
CONFIG_ESPTOOLPY_FLASHSIZE, 1771
CONFIG_FEATURE_CACHE_TX_BUF_BIT (C macro), 895
CONFIG_FEATURE_FTM_INITIATOR_BIT (C macro), 895
CONFIG_FEATURE_FTM_RESPONDER_BIT (C macro), 895
CONFIG_FEATURE_WPA3_SAE_BIT (C macro), 895
CONFIG_HEAP_TRACING_STACK_DEPTH (C macro), 2054

D

DAC_CHANNEL_1_GPIO_NUM (C macro), 1051
DAC_CHANNEL_2_GPIO_NUM (C macro), 1051
dac_channel_t (C++ enum), 1052
dac_channel_t::DAC_CHANNEL_1 (C++ enumerator), 1052
dac_channel_t::DAC_CHANNEL_2 (C++ enumerator), 1052
dac_channel_t::DAC_CHANNEL_MAX (C++ enumerator), 1052
dac_cw_config_t (C++ struct), 1051
dac_cw_config_t::en_ch (C++ member), 1051
dac_cw_config_t::freq (C++ member), 1051
dac_cw_config_t::offset (C++ member), 1051
dac_cw_config_t::phase (C++ member), 1051
dac_cw_config_t::scale (C++ member), 1051
dac_cw_generator_config (C++ function), 1050
dac_cw_generator_disable (C++ function), 1050
dac_cw_generator_enable (C++ function), 1050
dac_cw_phase_t (C++ enum), 1052
dac_cw_phase_t::DAC_CW_PHASE_0 (C++ enumerator), 1052
dac_cw_phase_t::DAC_CW_PHASE_180 (C++ enumerator), 1052
dac_cw_scale_t (C++ enum), 1052
dac_cw_scale_t::DAC_CW_SCALE_1 (C++ enumerator), 1052
dac_cw_scale_t::DAC_CW_SCALE_2 (C++ enumerator), 1052
dac_cw_scale_t::DAC_CW_SCALE_4 (C++ enumerator), 1052
dac_cw_scale_t::DAC_CW_SCALE_8 (C++ enumerator), 1052
DAC_GPIO25_CHANNEL (C macro), 1051
DAC_GPIO26_CHANNEL (C macro), 1051
dac_i2s_disable (C++ function), 1049
dac_i2s_enable (C++ function), 1049
dac_output_disable (C++ function), 1050
dac_output_enable (C++ function), 1050
dac_output_voltage (C++ function), 1050
dac_pad_get_io_num (C++ function), 1050
DEFAULT_HTTP_BUF_SIZE (C macro), 166
DEL_DEV_ADDR_FLAG (C macro), 579
DEL_DEV_UUID_FLAG (C macro), 579
dpp_bootstrap_type (C++ enum), 943
dpp_bootstrap_type::DPP_BOOTSTRAP_NFC_URI (C++ enumerator), 943
dpp_bootstrap_type::DPP_BOOTSTRAP_PKEX (C++ enumerator), 943
dpp_bootstrap_type::DPP_BOOTSTRAP_QR_CODE (C++ enumerator), 943

E

EFD_SUPPORT_ISR (C macro), 1823
efuse_hal_chip_revision (C++ function), 1840
efuse_hal_flash_encryption_enabled (C++ function), 1840
efuse_hal_get_mac (C++ function), 1840
efuse_hal_get_major_chip_version (C++ function), 1840
efuse_hal_get_minor_chip_version (C++ function), 1840
emac_rmii_clock_gpio_t (C++ enum), 968
emac_rmii_clock_gpio_t::EMAC_APPL_CLK_OUT_GPIO (C++ enumerator), 968
emac_rmii_clock_gpio_t::EMAC_CLK_IN_GPIO (C++ enumerator), 968

emac_rmii_clock_gpio_t::EMAC_CLK_OUT_180_GPIO (C++ enumerator), 969
emac_rmii_clock_gpio_t::EMAC_CLK_OUT_GPIO (C++ enumerator), 969
emac_rmii_clock_mode_t (C++ enum), 968
emac_rmii_clock_mode_t::EMAC_CLK_DEFAULT (C++ enumerator), 968
emac_rmii_clock_mode_t::EMAC_CLK_EXT_IN (C++ enumerator), 968
emac_rmii_clock_mode_t::EMAC_CLK_OUT (C++ enumerator), 968
eNotifyAction (C++ enum), 1934
eNotifyAction::eIncrement (C++ enumerator), 1934
eNotifyAction::eNoAction (C++ enumerator), 1934
eNotifyAction::eSetBits (C++ enumerator), 1934
eNotifyAction::eSetValueWithoutOverwrite (C++ enumerator), 1934
eNotifyAction::eSetValueWithOverwrite (C++ enumerator), 1934
environment variable CONFIG_ESPTOOLPY_FLASHSIZE, 1771
eSleepModeStatus (C++ enum), 1935
eSleepModeStatus::eAbortSleep (C++ enumerator), 1935
eSleepModeStatus::eNoTasksWaitingTimeout (C++ enumerator), 1935
eSleepModeStatus::eStandardSleep (C++ enumerator), 1935
esp_a2d_audio_state_t (C++ enum), 408
esp_a2d_audio_state_t::ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND (C++ enumerator), 408
esp_a2d_audio_state_t::ESP_A2D_AUDIO_STATE_STARTED (C++ enumerator), 408
esp_a2d_audio_state_t::ESP_A2D_AUDIO_STATE_STOPPED (C++ enumerator), 408
esp_a2d_cb_event_t (C++ enum), 409
esp_a2d_cb_event_t::ESP_A2D_AUDIO_CFG_EVT (C++ enumerator), 409
esp_a2d_cb_event_t::ESP_A2D_AUDIO_STATE_EVT (C++ enumerator), 409
esp_a2d_cb_event_t::ESP_A2D_CONNECTION_STATE_EVT (C++ enumerator), 409
esp_a2d_cb_event_t::ESP_A2D_MEDIA_CTRL_ACK_EVT (C++ enumerator), 409
esp_a2d_cb_event_t::ESP_A2D_PROF_STATE_EVT (C++ enumerator), 410
esp_a2d_cb_event_t::ESP_A2D_REPORT_SNK_DELAY_VALUE_EVT (C++ enumerator), 410
esp_a2d_cb_event_t::ESP_A2D_SNK_GET_DELAY_VALUE_EVT (C++ enumerator), 410
esp_a2d_cb_event_t::ESP_A2D_SNK_PSC_CFG_EVT (C++ enumerator), 410
esp_a2d_cb_event_t::ESP_A2D_SNK_SET_DELAY_VALUE_EVT (C++ enumerator), 410
esp_a2d_cb_param_t (C++ union), 403
esp_a2d_cb_param_t::a2d_audio_cfg_param (C++ struct), 404
esp_a2d_cb_param_t::a2d_audio_cfg_param::mcc (C++ member), 404
esp_a2d_cb_param_t::a2d_audio_cfg_param::remote_b (C++ member), 404
esp_a2d_cb_param_t::a2d_audio_stat_param (C++ struct), 404
esp_a2d_cb_param_t::a2d_audio_stat_param::remote_ (C++ member), 404
esp_a2d_cb_param_t::a2d_audio_stat_param::state (C++ member), 404
esp_a2d_cb_param_t::a2d_conn_stat_param (C++ struct), 404
esp_a2d_cb_param_t::a2d_conn_stat_param::disc_rsn (C++ member), 404
esp_a2d_cb_param_t::a2d_conn_stat_param::remote_b (C++ member), 404
esp_a2d_cb_param_t::a2d_conn_stat_param::state (C++ member), 404
esp_a2d_cb_param_t::a2d_get_delay_value_stat (C++ member), 404
esp_a2d_cb_param_t::a2d_get_stat_param (C++ struct), 404
esp_a2d_cb_param_t::a2d_get_stat_param::delay_val (C++ member), 405
esp_a2d_cb_param_t::a2d_prof_stat (C++ member), 403
esp_a2d_cb_param_t::a2d_prof_stat_param (C++ struct), 405
esp_a2d_cb_param_t::a2d_prof_stat_param::init_sta (C++ member), 405
esp_a2d_cb_param_t::a2d_psc_cfg_param (C++ struct), 405
esp_a2d_cb_param_t::a2d_psc_cfg_param::psc_mask (C++ member), 405
esp_a2d_cb_param_t::a2d_psc_cfg_stat (C++ member), 403
esp_a2d_cb_param_t::a2d_report_delay_stat_param (C++ struct), 405
esp_a2d_cb_param_t::a2d_report_delay_stat_param:: (C++ member), 405
esp_a2d_cb_param_t::a2d_report_delay_value_stat (C++ member), 404
esp_a2d_cb_param_t::a2d_set_delay_value_stat (C++ member), 403
esp_a2d_cb_param_t::a2d_set_stat_param (C++ struct), 405
esp_a2d_cb_param_t::a2d_set_stat_param::delay_val (C++ member), 405
esp_a2d_cb_param_t::a2d_set_stat_param::set_state (C++ member), 405
esp_a2d_cb_param_t::audio_cfg (C++ member), 403
esp_a2d_cb_param_t::audio_stat (C++ member), 403
esp_a2d_cb_param_t::conn_stat (C++ member), 403

esp_a2d_cb_param_t::media_ctrl_stat (C++ member), 403
esp_a2d_cb_param_t::media_ctrl_stat_param (C++ struct), 405
esp_a2d_cb_param_t::media_ctrl_stat_param::cmd (C++ member), 406
esp_a2d_cb_param_t::media_ctrl_stat_param::stat (C++ member), 406
esp_a2d_cb_t (C++ type), 407
ESP_A2D_CIE_LEN_ATRAC (C macro), 407
ESP_A2D_CIE_LEN_M12 (C macro), 407
ESP_A2D_CIE_LEN_M24 (C macro), 407
ESP_A2D_CIE_LEN_SBC (C macro), 407
esp_a2d_connection_state_t (C++ enum), 407
esp_a2d_connection_state_t::ESP_A2D_CONNECTION_STATE_CONNECTED (C++ enumerator), 408
esp_a2d_connection_state_t::ESP_A2D_CONNECTION_STATE_CONNECTING (C++ enumerator), 407
esp_a2d_connection_state_t::ESP_A2D_CONNECTION_STATE_DISCONNECTED (C++ enumerator), 407
esp_a2d_connection_state_t::ESP_A2D_CONNECTION_STATE_DISCONNECTING (C++ enumerator), 408
esp_a2d_disc_rsn_t (C++ enum), 408
esp_a2d_disc_rsn_t::ESP_A2D_DISC_RSN_ABNORMAL (C++ enumerator), 408
esp_a2d_disc_rsn_t::ESP_A2D_DISC_RSN_NORMAL (C++ enumerator), 408
esp_a2d_init_state_t (C++ enum), 409
esp_a2d_init_state_t::ESP_A2D_DEINIT_SUCCESS (C++ enumerator), 409
esp_a2d_init_state_t::ESP_A2D_INIT_SUCCESS (C++ enumerator), 409
esp_a2d_mcc_t (C++ struct), 406
esp_a2d_mcc_t::atrac (C++ member), 406
esp_a2d_mcc_t::cie (C++ member), 406
esp_a2d_mcc_t::m12 (C++ member), 406
esp_a2d_mcc_t::m24 (C++ member), 406
esp_a2d_mcc_t::sbc (C++ member), 406
esp_a2d_mcc_t::type (C++ member), 406
ESP_A2D_MCT_ATRAC (C macro), 406
ESP_A2D_MCT_M12 (C macro), 406
ESP_A2D_MCT_M24 (C macro), 406
ESP_A2D_MCT_NON_A2DP (C macro), 406
ESP_A2D_MCT_SBC (C macro), 406
esp_a2d_mct_t (C++ type), 407
esp_a2d_media_ctrl (C++ function), 402
esp_a2d_media_ctrl_ack_t (C++ enum), 408
esp_a2d_media_ctrl_ack_t::ESP_A2D_MEDIA_CTRL_ACK_BUSY (C++ enumerator), 408
esp_a2d_media_ctrl_ack_t::ESP_A2D_MEDIA_CTRL_ACK_FAILURE (C++ enumerator), 408
esp_a2d_media_ctrl_ack_t::ESP_A2D_MEDIA_CTRL_ACK_SUCCESS (C++ enumerator), 408
esp_a2d_media_ctrl_t (C++ enum), 408
esp_a2d_media_ctrl_t::ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY (C++ enumerator), 409
esp_a2d_media_ctrl_t::ESP_A2D_MEDIA_CTRL_NONE (C++ enumerator), 408
esp_a2d_media_ctrl_t::ESP_A2D_MEDIA_CTRL_START (C++ enumerator), 409
esp_a2d_media_ctrl_t::ESP_A2D_MEDIA_CTRL_STOP (C++ enumerator), 409
esp_a2d_media_ctrl_t::ESP_A2D_MEDIA_CTRL_SUSPEND (C++ enumerator), 409
ESP_A2D_PSC_DELAY_RPT (C macro), 407
esp_a2d_psc_t (C++ type), 407
esp_a2d_register_callback (C++ function), 401
esp_a2d_set_delay_value_state_t (C++ enum), 409
esp_a2d_set_delay_value_state_t::ESP_A2D_SET_INVALID_PARAM (C++ enumerator), 409
esp_a2d_set_delay_value_state_t::ESP_A2D_SET_SUCCESS (C++ enumerator), 409
esp_a2d_sink_connect (C++ function), 401
esp_a2d_sink_data_cb_t (C++ type), 407
esp_a2d_sink_deinit (C++ function), 401
esp_a2d_sink_disconnect (C++ function), 401
esp_a2d_sink_get_delay_value (C++ function), 402
esp_a2d_sink_init (C++ function), 401
esp_a2d_sink_register_data_callback (C++ function), 401
esp_a2d_sink_set_delay_value (C++ function), 402
esp_a2d_source_connect (C++ function), 403
esp_a2d_source_data_cb_t (C++ type), 407
esp_a2d_source_deinit (C++ function), 402
esp_a2d_source_disconnect (C++ function), 403
esp_a2d_source_init (C++ function), 402
esp_a2d_source_register_data_callback (C++ function), 402
esp_alloc_failed_hook_t (C++ type), 2039
ESP_APP_DESC_MAGIC_WORD (C macro), 2095
esp_app_desc_t (C++ struct), 2095
esp_app_desc_t::app_elf_sha256 (C++ member), 2095
esp_app_desc_t::date (C++ member), 2095
esp_app_desc_t::idf_ver (C++ member), 2095
esp_app_desc_t::magic_word (C++ member), 2095
esp_app_desc_t::project_name (C++ member), 2095
esp_app_desc_t::reserv1 (C++ member), 2095
esp_app_desc_t::reserv2 (C++ member), 2095
esp_app_desc_t::secure_version (C++ member), 2095
esp_app_desc_t::time (C++ member), 2095
esp_app_desc_t::version (C++ member), 2095
esp_app_get_description (C++ function),

2094
esp_app_get_elf_sha256 (C++ function), 2094
ESP_APP_ID_MAX (C macro), 238
ESP_APP_ID_MIN (C macro), 238
esp_apptrace_buffer_get (C++ function), 1832
esp_apptrace_buffer_put (C++ function), 1832
esp_apptrace_dest_t (C++ enum), 1835
esp_apptrace_dest_t::ESP_APPTRACE_DEST_JTAG (C++ enumerator), 1835
esp_apptrace_dest_t::ESP_APPTRACE_DEST_MAX (C++ enumerator), 1835
esp_apptrace_dest_t::ESP_APPTRACE_DEST_NUM (C++ enumerator), 1835
esp_apptrace_dest_t::ESP_APPTRACE_DEST_TRAX (C++ enumerator), 1835
esp_apptrace_dest_t::ESP_APPTRACE_DEST_UART (C++ enumerator), 1835
esp_apptrace_down_buffer_config (C++ function), 1832
esp_apptrace_down_buffer_get (C++ function), 1833
esp_apptrace_down_buffer_put (C++ function), 1834
esp_apptrace_fclose (C++ function), 1834
esp_apptrace_flush (C++ function), 1833
esp_apptrace_flush_nolock (C++ function), 1833
esp_apptrace_fopen (C++ function), 1834
esp_apptrace_fread (C++ function), 1834
esp_apptrace_fseek (C++ function), 1834
esp_apptrace_fstop (C++ function), 1835
esp_apptrace_ftell (C++ function), 1835
esp_apptrace_fwrite (C++ function), 1834
esp_apptrace_host_is_connected (C++ function), 1834
esp_apptrace_init (C++ function), 1832
esp_apptrace_read (C++ function), 1833
esp_apptrace_vprintf (C++ function), 1833
esp_apptrace_vprintf_to (C++ function), 1833
esp_apptrace_write (C++ function), 1832
esp_attr_control_t (C++ struct), 318
esp_attr_control_t::auto_rsp (C++ member), 318
esp_attr_desc_t (C++ struct), 317
esp_attr_desc_t::length (C++ member), 318
esp_attr_desc_t::max_length (C++ member), 317
esp_attr_desc_t::perm (C++ member), 317
esp_attr_desc_t::uuid_length (C++ member), 317
esp_attr_desc_t::uuid_p (C++ member), 317
esp_attr_desc_t::value (C++ member), 318
esp_attr_value_t (C++ struct), 318
esp_attr_value_t::attr_len (C++ member), 318
esp_attr_value_t::attr_max_len (C++ member), 318
esp_attr_value_t::attr_value (C++ member), 318
esp_avrc_batt_stat_t (C++ enum), 430
esp_avrc_batt_stat_t::ESP_AVRC_BATT_CRITICAL (C++ enumerator), 431
esp_avrc_batt_stat_t::ESP_AVRC_BATT_EXTERNAL (C++ enumerator), 431
esp_avrc_batt_stat_t::ESP_AVRC_BATT_FULL_CHARGE (C++ enumerator), 431
esp_avrc_batt_stat_t::ESP_AVRC_BATT_NORMAL (C++ enumerator), 430
esp_avrc_batt_stat_t::ESP_AVRC_BATT_WARNING (C++ enumerator), 430
esp_avrc_bit_mask_op_t (C++ enum), 425
esp_avrc_bit_mask_op_t::ESP_AVRC_BIT_MASK_OP_CLEA (C++ enumerator), 425
esp_avrc_bit_mask_op_t::ESP_AVRC_BIT_MASK_OP_SET (C++ enumerator), 425
esp_avrc_bit_mask_op_t::ESP_AVRC_BIT_MASK_OP_TEST (C++ enumerator), 425
esp_avrc_ct_cb_event_t (C++ enum), 425
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_CHANGE_NOTIFY (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_CONNECTION_ST (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_GET_RN_CAPABI (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_METADATA_RSP_ (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_PASSTHROUGH_R (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_PLAY_STATUS_R (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_REMOTE_FEATUR (C++ enumerator), 426
esp_avrc_ct_cb_event_t::ESP_AVRC_CT_SET_ABSOLUTE_ (C++ enumerator), 426
esp_avrc_ct_cb_param_t (C++ union), 415
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_par (C++ struct), 415
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_par (C++ member), 415
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_par (C++ member), 415
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (C++ struct), 416
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param:: (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param:: (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_p (C++ struct), 416
esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_p (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_p (C++ member), 416

esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ struct), 416
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ struct), 416
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 417
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 417
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 416
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param (C++ struct), 417
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param (C++ member), 417
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param (C++ member), 417
esp_avrc_ct_cb_param_t::avrc_ct_set_volume_rsp_param (C++ struct), 417
esp_avrc_ct_cb_param_t::avrc_ct_set_volume_rsp_param (C++ member), 417
esp_avrc_ct_cb_param_t::change_ntf (C++ member), 415
esp_avrc_ct_cb_param_t::conn_stat (C++ member), 415
esp_avrc_ct_cb_param_t::get_rn_caps_rsp (C++ member), 415
esp_avrc_ct_cb_param_t::meta_rsp (C++ member), 415
esp_avrc_ct_cb_param_t::psth_rsp (C++ member), 415
esp_avrc_ct_cb_param_t::rmt_feats (C++ member), 415
esp_avrc_ct_cb_param_t::set_volume_rsp (C++ member), 415
esp_avrc_ct_cb_t (C++ type), 420
esp_avrc_ct_deinit (C++ function), 410
esp_avrc_ct_init (C++ function), 410
esp_avrc_ct_register_callback (C++ function), 410
esp_avrc_ct_send_get_rn_capabilities_cmd (C++ function), 411
esp_avrc_ct_send_metadata_cmd (C++ function), 412
esp_avrc_ct_send_passthrough_cmd (C++ function), 412
esp_avrc_ct_send_register_notification_cmd (C++ function), 411
esp_avrc_ct_send_set_absolute_volume_cmd (C++ function), 411
esp_avrc_ct_send_set_player_value_cmd (C++ function), 411
esp_avrc_feature_flag_t (C++ enum), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_BROWSE (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_CAT1 (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_CAT2 (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_CAT3 (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_CAT4 (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE_FEATURE (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_COVER_ART_GET_LINKED_THUMBNAIL (C++ enumerator), 421
esp_avrc_feature_flag_t::ESP_AVRC_FEAT_FLAG_COVER_ART_PUT_IMAGE (C++ enumerator), 421
esp_avrc_features_t (C++ enum), 420
esp_avrc_features_t::ESP_AVRC_FEAT_ADV_CTRL (C++ enumerator), 420
esp_avrc_features_t::ESP_AVRC_FEAT_BROWSE (C++ enumerator), 420
esp_avrc_features_t::ESP_AVRC_FEAT_META_DATA (C++ enumerator), 420
esp_avrc_features_t::ESP_AVRC_FEAT_RCCT (C++ enumerator), 420
esp_avrc_features_t::ESP_AVRC_FEAT_RCTG (C++ enumerator), 420
esp_avrc_features_t::ESP_AVRC_FEAT_VENDOR (C++ enumerator), 420
esp_avrc_md_attr_mask_t (C++ enum), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_ALBUM (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_ARTIST (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_GENRE (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_NUM_TRA (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_PLAYING (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_TITLE (C++ enumerator), 427
esp_avrc_md_attr_mask_t::ESP_AVRC_MD_ATTR_TRACK_N (C++ enumerator), 427
esp_avrc_playback_stat_t (C++ enum), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_ERROR (C++ enumerator), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_FWD_S (C++ enumerator), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_PAUSE (C++ enumerator), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_PLAYI (C++ enumerator), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_REV_S

(C++ enumerator), 431
esp_avrc_playback_stat_t::ESP_AVRC_PLAYBACK_STOPPED (C++ enumerator), 431
esp_avrc_ps_attr_ids_t (C++ enum), 428
esp_avrc_ps_attr_ids_t::ESP_AVRC_PS_EQUALIZER (C++ enumerator), 429
esp_avrc_ps_attr_ids_t::ESP_AVRC_PS_MAX_ATTR (C++ enumerator), 429
esp_avrc_ps_attr_ids_t::ESP_AVRC_PS_REPEAT_MODE (C++ enumerator), 429
esp_avrc_ps_attr_ids_t::ESP_AVRC_PS_SCAN_MODE (C++ enumerator), 429
esp_avrc_ps_attr_ids_t::ESP_AVRC_PS_SHUFFLE_MODE (C++ enumerator), 429
esp_avrc_ps_eq_value_ids_t (C++ enum), 429
esp_avrc_ps_eq_value_ids_t::ESP_AVRC_PS_EQUALIZER_OFF (C++ enumerator), 429
esp_avrc_ps_eq_value_ids_t::ESP_AVRC_PS_EQUALIZER_ON (C++ enumerator), 429
esp_avrc_ps_rpt_value_ids_t (C++ enum), 429
esp_avrc_ps_rpt_value_ids_t::ESP_AVRC_PS_REPEAT_GROUP (C++ enumerator), 429
esp_avrc_ps_rpt_value_ids_t::ESP_AVRC_PS_REPEAT_OFF (C++ enumerator), 429
esp_avrc_ps_rpt_value_ids_t::ESP_AVRC_PS_REPEAT_SINGLE (C++ enumerator), 429
esp_avrc_ps_scn_value_ids_t (C++ enum), 430
esp_avrc_ps_scn_value_ids_t::ESP_AVRC_PS_SCAN_ALL (C++ enumerator), 430
esp_avrc_ps_scn_value_ids_t::ESP_AVRC_PS_SCAN_GROUP (C++ enumerator), 430
esp_avrc_ps_scn_value_ids_t::ESP_AVRC_PS_SCAN_OFF (C++ enumerator), 430
esp_avrc_ps_shf_value_ids_t (C++ enum), 429
esp_avrc_ps_shf_value_ids_t::ESP_AVRC_PS_SHUFFLE_OFF_ALL (C++ enumerator), 429
esp_avrc_ps_shf_value_ids_t::ESP_AVRC_PS_SHUFFLE_OFF_GROUP (C++ enumerator), 430
esp_avrc_ps_shf_value_ids_t::ESP_AVRC_PS_SHUFFLE_OFF (C++ enumerator), 429
esp_avrc_psth_bit_mask_operation (C++ function), 413
esp_avrc_psth_bit_mask_t (C++ struct), 419
esp_avrc_psth_bit_mask_t::bits (C++ member), 419
esp_avrc_psth_filter_t (C++ enum), 425
esp_avrc_psth_filter_t::ESP_AVRC_PSTH_FILTER_ALLOWED_CMD (C++ enumerator), 425
esp_avrc_psth_filter_t::ESP_AVRC_PSTH_FILTER_SUPPORTED_CMD (C++ enumerator), 425
esp_avrc_pt_cmd_state_t (C++ enum), 425
esp_avrc_pt_cmd_state_t::ESP_AVRC_PT_CMD_STATE_PRESSED (C++ enumerator), 425
esp_avrc_pt_cmd_state_t::ESP_AVRC_PT_CMD_STATE_RELEASED (C++ enumerator), 425
esp_avrc_pt_cmd_t (C++ enum), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_0 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_1 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_2 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_3 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_4 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_5 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_6 (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_7 (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_8 (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_9 (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_ANGLE (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_BACKWARD (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_CHAN_DOWN (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_CHAN_UP (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_CLEAR (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_CONT_MENU (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_DISP_INFO (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_DOT (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_DOWN (C++ enumerator), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_EJECT (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_ENTER (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_EXIT (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_F1 (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_F2 (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_F3 (C++ enumerator), 425
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_F4 (C++ enumerator), 425
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_F5 (C++ enumerator), 425

esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_FAST_FORWARD (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_FAV_MENU (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_FORWARD (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_HELP (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_INPUT_SEL (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_LEFT (C++ enumerator), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_LEFT_DOWN (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_LEFT_UP (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_MUTE (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_PAGE_DOWN (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_PAGE_UP (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_PAUSE (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_PLAY (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_POWER (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_PREV_CHAN (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_RECORD (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_REWIND (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_RIGHT (C++ enumerator), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_RIGHT_DOWN (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_RIGHT_UP (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_ROOT_MENU (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_SELECT (C++ enumerator), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_SETUP_MENU (C++ enumerator), 422
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_SOUND_SEL (C++ enumerator), 423
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_STOP (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_SUBPICT (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_UP (C++ enumerator), 421
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_VENDOR (C++ enumerator), 425
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_VOL_DOWN (C++ enumerator), 424
esp_avrc_pt_cmd_t::ESP_AVRC_PT_CMD_VOL_UP (C++ enumerator), 424
esp_avrc_rn_event_ids_t (C++ enum), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_ADDRESSED_PL (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_APP_SETTING_ (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_AVAILABLE_PL (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_BATTERY_STAT (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_MAX_EVT (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_NOW_PLAYING_ (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_PLAY_POS_CHA (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_PLAY_STATUS_ (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_SYSTEM_STATU (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_TRACK_CHANGE (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_TRACK_REACHE (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_TRACK_REACHE (C++ enumerator), 427
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_UIDS_CHANGE (C++ enumerator), 428
esp_avrc_rn_event_ids_t::ESP_AVRC_RN_VOLUME_CHANG (C++ enumerator), 428
esp_avrc_rn_evt_bit_mask_operation (C++ function), 414
esp_avrc_rn_evt_cap_mask_t (C++ struct), 419
esp_avrc_rn_evt_cap_mask_t::bits (C++ member), 419
esp_avrc_rn_evt_cap_t (C++ enum), 428
esp_avrc_rn_evt_cap_t::ESP_AVRC_RN_CAP_ALLOWED_EV (C++ enumerator), 428
esp_avrc_rn_evt_cap_t::ESP_AVRC_RN_CAP_MAX (C++ enumerator), 428
esp_avrc_rn_evt_cap_t::ESP_AVRC_RN_CAP_SUPPORTED_ (C++ enumerator), 428
esp_avrc_rn_param_t (C++ union), 414
esp_avrc_rn_param_t::batt (C++ member), 415
esp_avrc_rn_param_t::elm_id (C++ member), 415
esp_avrc_rn_param_t::play_pos (C++ member), 415
esp_avrc_rn_param_t::playback (C++ member), 415
esp_avrc_rn_param_t::volume (C++ member), 414
esp_avrc_rn_rsp_t (C++ enum), 428
esp_avrc_rn_rsp_t::ESP_AVRC_RN_RSP_CHANGED (C++ enumerator), 428

esp_avrc_rn_rsp_t::ESP_AVRC_RN_RSP_INTERIM (C++ enumerator), 428
esp_avrc_rsp_t (C++ enum), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_ACCEPT (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_CHANGED (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_IMPL_STBL (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_IN_TRANS (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_INTERIM (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_NOT_IMPL (C++ enumerator), 430
esp_avrc_rsp_t::ESP_AVRC_RSP_REJECT (C++ enumerator), 430
esp_avrc_set_app_value_param_t (C++ struct), 420
esp_avrc_set_app_value_param_t::attr_id (C++ member), 420
esp_avrc_set_app_value_param_t::attr_val (C++ member), 420
esp_avrc_tg_cb_event_t (C++ enum), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_CONNECTION_STATE_EVT (C++ enumerator), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_PASSTHROUGH_CMD_EVT (C++ enumerator), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT (C++ enumerator), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_REMOTE_FEATURES_EVT (C++ enumerator), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT (C++ enumerator), 426
esp_avrc_tg_cb_event_t::ESP_AVRC_TG_SET_PLAYER_APP_VALUE_EVT (C++ enumerator), 426
esp_avrc_tg_cb_param_t (C++ union), 417
esp_avrc_tg_cb_param_t::avrc_tg_conn_stat (C++ struct), 418
esp_avrc_tg_cb_param_t::avrc_tg_conn_stat::connected (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_conn_stat::remote_bda (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd (C++ struct), 418
esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd::key_code (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd::key_state (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf (C++ struct), 418
esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf::event_id (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf::param (C++ member), 418
esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats (C++ struct), 418
esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats_param::feat_mask (C++ member), 419
esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats_param::tg_feat_flag (C++ member), 419
esp_avrc_tg_cb_param_t::avrc_tg_set_abs_vol_param (C++ struct), 419
esp_avrc_tg_cb_param_t::avrc_tg_set_abs_vol_param::volume (C++ member), 419
esp_avrc_tg_cb_param_t::avrc_tg_set_app_value_par (C++ struct), 419
esp_avrc_tg_cb_param_t::avrc_tg_set_app_value_par::p_vals (C++ member), 419
esp_avrc_tg_cb_param_t::avrc_tg_set_app_value_par::p_vals (C++ member), 419
esp_avrc_tg_cb_param_t::conn_stat (C++ member), 417
esp_avrc_tg_cb_param_t::psth_cmd (C++ member), 417
esp_avrc_tg_cb_param_t::reg_ntf (C++ member), 418
esp_avrc_tg_cb_param_t::rmt_feats (C++ member), 417
esp_avrc_tg_cb_param_t::set_abs_vol (C++ member), 417
esp_avrc_tg_cb_param_t::set_app_value (C++ member), 418
esp_avrc_tg_cb_t (C++ type), 420
esp_avrc_tg_get_psth_cmd_filter (C++ function), 412
esp_avrc_tg_get_psth_cmd_filter (C++ function), 412
esp_avrc_tg_get_rn_evt_cap (C++ function), 413
esp_avrc_tg_init (C++ function), 412
esp_avrc_tg_register_callback (C++ function), 412
esp_avrc_tg_send_rn_rsp (C++ function), 414
esp_avrc_tg_set_psth_cmd_filter (C++ function), 413
esp_avrc_tg_set_rn_evt_cap (C++ function), 414
ESP_AVRC_TRANS_LABEL_MAX (C macro), 420
esp_base_mac_addr_get (C++ function), 2087
esp_base_mac_addr_set (C++ function), 2086
ESP_BD_ADDR_HEX (C macro), 238
ESP_BD_ADDR_LEN (C macro), 238
ESP_BD_ADDR_STR (C macro), 238
esp_bd_addr_t (C++ type), 238
esp_ble_addr_type_t (C++ enum), 242
esp_ble_addr_type_t::BLE_ADDR_TYPE_PUBLIC (C++ enumerator), 243
esp_ble_addr_type_t::BLE_ADDR_TYPE_RANDOM (C++ enumerator), 243
esp_ble_addr_type_t::BLE_ADDR_TYPE_RPA_PUBLIC (C++ enumerator), 243
esp_ble_addr_type_t::BLE_ADDR_TYPE_RPA_RANDOM (C++ enumerator), 243
esp_ble_adv_channel_t (C++ enum), 310

esp_ble_adv_channel_t::ADV_CHNL_37 (*C++ enumerator*), 311
esp_ble_adv_channel_t::ADV_CHNL_38 (*C++ enumerator*), 311
esp_ble_adv_channel_t::ADV_CHNL_39 (*C++ enumerator*), 311
esp_ble_adv_channel_t::ADV_CHNL_ALL (*C++ enumerator*), 311
ESP_BLE_ADV_DATA_LEN_MAX (*C macro*), 299
esp_ble_adv_data_t (*C++ struct*), 279
esp_ble_adv_data_t::appearance (*C++ member*), 280
esp_ble_adv_data_t::flag (*C++ member*), 280
esp_ble_adv_data_t::include_name (*C++ member*), 280
esp_ble_adv_data_t::include_txpower (*C++ member*), 280
esp_ble_adv_data_t::manufacturer_len (*C++ member*), 280
esp_ble_adv_data_t::max_interval (*C++ member*), 280
esp_ble_adv_data_t::min_interval (*C++ member*), 280
esp_ble_adv_data_t::p_manufacturer_data (*C++ member*), 280
esp_ble_adv_data_t::p_service_data (*C++ member*), 280
esp_ble_adv_data_t::p_service_uuid (*C++ member*), 280
esp_ble_adv_data_t::service_data_len (*C++ member*), 280
esp_ble_adv_data_t::service_uuid_len (*C++ member*), 280
esp_ble_adv_data_t::set_scan_rsp (*C++ member*), 280
esp_ble_adv_data_type (*C++ enum*), 309
esp_ble_adv_data_type::ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_16SRV_CMPL (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_16SRV_PART (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_32SOL_SRV_UUID (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_32SRV_CMPL (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_32SRV_PART (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_32SRV_UUID (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_ADV_INT (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_APPEARANCE (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_CHAN_MAP_U (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_DEV_CLASS (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_FLAG (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_INDOOR_POS (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_INT_RANGE (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_LE_DEV_ADD (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_LE_ROLE (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_LE_SECURE_ (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_LE_SECURE_ (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_LE_SUPPORT (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_NAME_CMPL (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_NAME_SHORT (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_PUBLIC_TAR (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_RANDOM_TAR (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SERVICE_DA (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SM_OOB_FLA (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SM_TK (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SOL_SRV_UU (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SPAIR_C256 (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_SPAIR_R256 (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_TRANS_DISC (*C++ enumerator*), 310
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_TX_PWR (*C++ enumerator*), 309
esp_ble_adv_data_type::ESP_BLE_AD_TYPE_URI (*C++ enumerator*), 310
esp_ble_adv_filter_t (*C++ enum*), 311
esp_ble_adv_filter_t::ADV_FILTER_ALLOW_SCAN_ANY_C (*C++ enumerator*), 311
esp_ble_adv_filter_t::ADV_FILTER_ALLOW_SCAN_ANY_C (*C++ enumerator*), 311
esp_ble_adv_filter_t::ADV_FILTER_ALLOW_SCAN_WLST_ (*C++ enumerator*), 311

esp_ble_adv_filter_t::ADV_FILTER_ALLOW_SCAN_WLST (C++ enumerator), 311
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT (C macro), 293
ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT (C macro), 293
ESP_BLE_ADV_FLAG_DMT_HOST_SPT (C macro), 293
ESP_BLE_ADV_FLAG_GEN_DISC (C macro), 293
ESP_BLE_ADV_FLAG_LIMIT_DISC (C macro), 293
ESP_BLE_ADV_FLAG_NON_LIMIT_DISC (C macro), 293
esp_ble_adv_params_t (C++ struct), 279
esp_ble_adv_params_t::adv_filter_policy (C++ member), 279
esp_ble_adv_params_t::adv_int_max (C++ member), 279
esp_ble_adv_params_t::adv_int_min (C++ member), 279
esp_ble_adv_params_t::adv_type (C++ member), 279
esp_ble_adv_params_t::channel_map (C++ member), 279
esp_ble_adv_params_t::own_addr_type (C++ member), 279
esp_ble_adv_params_t::peer_addr (C++ member), 279
esp_ble_adv_params_t::peer_addr_type (C++ member), 279
ESP_BLE_ADV_REPORT_EXT_ADV_IND (C macro), 302
ESP_BLE_ADV_REPORT_EXT_DIRECT_ADV (C macro), 302
ESP_BLE_ADV_REPORT_EXT_SCAN_IND (C macro), 302
ESP_BLE_ADV_REPORT_EXT_SCAN_RSP (C macro), 302
esp_ble_adv_type_t (C++ enum), 310
esp_ble_adv_type_t::ADV_TYPE_DIRECT_IND_HIGH (C++ enumerator), 310
esp_ble_adv_type_t::ADV_TYPE_DIRECT_IND_LOW (C++ enumerator), 310
esp_ble_adv_type_t::ADV_TYPE_IND (C++ enumerator), 310
esp_ble_adv_type_t::ADV_TYPE_NONCONN_IND (C++ enumerator), 310
esp_ble_adv_type_t::ADV_TYPE_SCAN_IND (C++ enumerator), 310
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM (C macro), 296
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST (C macro), 296
ESP_BLE_APPEARANCE_CYCLING_CADENCE (C macro), 297
ESP_BLE_APPEARANCE_CYCLING_COMPUTER (C macro), 297
ESP_BLE_APPEARANCE_CYCLING_POWER (C macro), 297
ESP_BLE_APPEARANCE_CYCLING_SPEED (C macro), 297
ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE (C macro), 297
ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_CLOCK (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_COMPUTER (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MON (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_CYCLING (C macro), 297
ESP_BLE_APPEARANCE_GENERIC_DISPLAY (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_GLUCOSE (C macro), 297
ESP_BLE_APPEARANCE_GENERIC_HEART_RATE (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_HID (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_KEYRING (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVI (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_PHONE (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER (C macro), 298
ESP_BLE_APPEARANCE_GENERIC_REMOTE (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_TAG (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_THERMOMETER (C macro), 296
ESP_BLE_APPEARANCE_GENERIC_WALKING (C macro), 297
ESP_BLE_APPEARANCE_GENERIC_WATCH (C macro), 295
ESP_BLE_APPEARANCE_GENERIC_WEIGHT (C macro), 298
ESP_BLE_APPEARANCE_HEART_RATE_BELT (C macro), 296
ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER

(*C macro*), 297
ESP_BLE_APPEARANCE_HID_CARD_READER (*C macro*), 297
ESP_BLE_APPEARANCE_HID_DIGITAL_PEN (*C macro*), 297
ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET (*C macro*), 297
ESP_BLE_APPEARANCE_HID_GAMEPAD (*C macro*), 296
ESP_BLE_APPEARANCE_HID_JOYSTICK (*C macro*), 296
ESP_BLE_APPEARANCE_HID_KEYBOARD (*C macro*), 296
ESP_BLE_APPEARANCE_HID_MOUSE (*C macro*), 296
ESP_BLE_APPEARANCE_INSULIN_PEN (*C macro*), 298
ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP (*C macro*), 298
ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP (*C macro*), 298
ESP_BLE_APPEARANCE_MOBILITY_SCOOTER (*C macro*), 298
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 298
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV (*C macro*), 298
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD (*C macro*), 299
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV (*C macro*), 299
ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR (*C macro*), 298
ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP (*C macro*), 298
ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST (*C macro*), 298
ESP_BLE_APPEARANCE_SPORTS_WATCH (*C macro*), 295
ESP_BLE_APPEARANCE_STANDALONE_SPEAKER (*C macro*), 297
ESP_BLE_APPEARANCE_THERMOMETER_EAR (*C macro*), 296
ESP_BLE_APPEARANCE_UNKNOWN (*C macro*), 295
ESP_BLE_APPEARANCE_WALKING_IN_SHOE (*C macro*), 297
ESP_BLE_APPEARANCE_WALKING_ON_HIP (*C macro*), 297
ESP_BLE_APPEARANCE_WALKING_ON_SHOE (*C macro*), 297
esp_ble_auth_cmpl_t (*C++ struct*), 286
esp_ble_auth_cmpl_t::addr_type (*C++ member*), 286
esp_ble_auth_cmpl_t::auth_mode (*C++ member*), 286
esp_ble_auth_cmpl_t::bd_addr (*C++ member*), 286
esp_ble_auth_cmpl_t::dev_type (*C++ member*), 286
esp_ble_auth_cmpl_t::fail_reason (*C++ member*), 286
esp_ble_auth_cmpl_t::key (*C++ member*), 286
esp_ble_auth_cmpl_t::key_present (*C++ member*), 286
esp_ble_auth_cmpl_t::key_type (*C++ member*), 286
esp_ble_auth_cmpl_t::success (*C++ member*), 286
esp_ble_auth_req_t (*C++ type*), 303
esp_ble_bond_dev_t (*C++ struct*), 285
esp_ble_bond_dev_t::bd_addr (*C++ member*), 285
esp_ble_bond_dev_t::bond_key (*C++ member*), 285
esp_ble_bond_key_info_t (*C++ struct*), 284
esp_ble_bond_key_info_t::key_mask (*C++ member*), 284
esp_ble_bond_key_info_t::pcsrk_key (*C++ member*), 284
esp_ble_bond_key_info_t::penc_key (*C++ member*), 284
esp_ble_bond_key_info_t::pid_key (*C++ member*), 284
esp_ble_confirm_reply (*C++ function*), 250
ESP_BLE_CONN_INT_MAX (*C macro*), 237
ESP_BLE_CONN_INT_MIN (*C macro*), 237
ESP_BLE_CONN_LATENCY_MAX (*C macro*), 237
ESP_BLE_CONN_SUP_TOUT_MAX (*C macro*), 237
ESP_BLE_CONN_SUP_TOUT_MIN (*C macro*), 237
esp_ble_conn_update_params_t (*C++ struct*), 281
esp_ble_conn_update_params_t::bda (*C++ member*), 281
esp_ble_conn_update_params_t::latency (*C++ member*), 282
esp_ble_conn_update_params_t::max_int (*C++ member*), 282
esp_ble_conn_update_params_t::min_int (*C++ member*), 281
esp_ble_conn_update_params_t::timeout (*C++ member*), 282
esp_ble_create_sc_oob_data (*C++ function*), 251
ESP_BLE_CSR_KEY_MASK (*C macro*), 238
esp_ble_dtm_enh_rx_start (*C++ function*), 257
esp_ble_dtm_enh_rx_t (*C++ struct*), 292
esp_ble_dtm_enh_rx_t::modulation_idx (*C++ member*), 293
esp_ble_dtm_enh_rx_t::phy (*C++ member*), 293
esp_ble_dtm_enh_rx_t::rx_channel (*C++ member*), 292
esp_ble_dtm_enh_tx_start (*C++ function*), 257

esp_ble_dtm_enh_tx_t (*C++ struct*), 292
esp_ble_dtm_enh_tx_t::len_of_data (*C++ member*), 292
esp_ble_dtm_enh_tx_t::phy (*C++ member*), 292
esp_ble_dtm_enh_tx_t::pkt_payload (*C++ member*), 292
esp_ble_dtm_enh_tx_t::tx_channel (*C++ member*), 292
esp_ble_dtm_pkt_payload_t (*C++ type*), 303
esp_ble_dtm_rx_start (*C++ function*), 257
esp_ble_dtm_rx_t (*C++ struct*), 279
esp_ble_dtm_rx_t::rx_channel (*C++ member*), 279
esp_ble_dtm_stop (*C++ function*), 257
esp_ble_dtm_tx_start (*C++ function*), 257
esp_ble_dtm_tx_t (*C++ struct*), 278
esp_ble_dtm_tx_t::len_of_data (*C++ member*), 278
esp_ble_dtm_tx_t::pkt_payload (*C++ member*), 279
esp_ble_dtm_tx_t::tx_channel (*C++ member*), 278
esp_ble_dtm_update_evt_t (*C++ enum*), 312
esp_ble_dtm_update_evt_t::DTM_RX_START_EVT (*C++ enumerator*), 312
esp_ble_dtm_update_evt_t::DTM_TEST_STOP_EVT (*C++ enumerator*), 312
esp_ble_dtm_update_evt_t::DTM_TX_START_EVT (*C++ enumerator*), 312
esp_ble_duplicate_exceptional_info_type_t (*C++ enum*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_ADV_A (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 315
esp_ble_duplicate_exceptional_info_type_t::ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_ (*C++ enumerator*), 316
ESP_BLE_ENC_KEY_MASK (*C macro*), 238
esp_ble_evt_type_t (*C++ enum*), 314
esp_ble_evt_type_t::ESP_BLE_EVT_CONN_ADV (*C++ enumerator*), 314
esp_ble_evt_type_t::ESP_BLE_EVT_CONN_DIR_ADV (*C++ enumerator*), 314
esp_ble_evt_type_t::ESP_BLE_EVT_DISC_ADV (*C++ enumerator*), 314
esp_ble_evt_type_t::ESP_BLE_EVT_NON_CONN_ADV (*C++ enumerator*), 314
esp_ble_evt_type_t::ESP_BLE_EVT_SCAN_RSP (*C++ enumerator*), 315
esp_ble_ext_adv_type_mask_t (*C++ type*), 303
esp_ble_ext_scan_cfg_mask_t (*C++ type*), 303
esp_ble_ext_scan_cfg_t (*C++ struct*), 287
esp_ble_ext_scan_cfg_t::scan_interval (*C++ member*), 287
esp_ble_ext_scan_cfg_t::scan_type (*C++ member*), 287
esp_ble_ext_scan_cfg_t::scan_window (*C++ member*), 288
esp_ble_ext_scan_params_t (*C++ struct*), 288
esp_ble_ext_scan_params_t::cfg_mask (*C++ member*), 288
esp_ble_ext_scan_params_t::coded_cfg (*C++ member*), 288
esp_ble_ext_scan_params_t::filter_policy (*C++ member*), 288
esp_ble_ext_scan_params_t::own_addr_type (*C++ member*), 288
esp_ble_ext_scan_params_t::scan_duplicate (*C++ member*), 288
esp_ble_ext_scan_params_t::uncoded_cfg (*C++ member*), 288
ESP_BLE_GAP_ADD_DUPLICATE_SCAN_EXCEPTIONAL_DEVICE (*C++ function*), 249
esp_ble_gap_adv_type_t (*C++ type*), 303
esp_ble_gap_all_phys_t (*C++ type*), 303
esp_ble_gap_cb_param_t (*C++ union*), 258
esp_ble_gap_cb_param_t::adv_clear_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::adv_data_cmpl (*C++ member*), 258
esp_ble_gap_cb_param_t::adv_data_raw_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::adv_start_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::adv_stop_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::adv_terminate (*C++ member*), 261
esp_ble_gap_cb_param_t::ble_adv_clear_cmpl_evt_pa (*C++ struct*), 262
esp_ble_gap_cb_param_t::ble_adv_clear_cmpl_evt_pa (*C++ member*), 263
esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_par (*C++ struct*), 263
esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_par (*C++ member*), 263
esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt (*C++ struct*), 263
esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt (*C++ member*), 263
esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_pa (*C++ struct*), 263
esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_pa (*C++ member*), 263
esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_par

(C++ *struct*), 263
esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_param (C++ *member*), 263
esp_ble_gap_cb_param_t::ble_adv_terminate_param (C++ *struct*), 263
esp_ble_gap_cb_param_t::ble_adv_terminate_param::adv_instance (C++ *member*), 263
esp_ble_gap_cb_param_t::ble_adv_terminate_param::conn_idx (C++ *member*), 264
esp_ble_gap_cb_param_t::ble_adv_terminate_param::status (C++ *member*), 263
esp_ble_gap_cb_param_t::ble_channel_select_alg_param (C++ *struct*), 264
esp_ble_gap_cb_param_t::ble_channel_select_alg_param::channel_selection_algorithm (C++ *member*), 264
esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_param (C++ *struct*), 264
esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_param::status (C++ *member*), 264
esp_ble_gap_cb_param_t::ble_dtm_state_update_param (C++ *struct*), 264
esp_ble_gap_cb_param_t::ble_dtm_state_update_param::num_of_packet (C++ *member*), 264
esp_ble_gap_cb_param_t::ble_dtm_state_update_param::status (C++ *member*), 264
esp_ble_gap_cb_param_t::ble_ext_adv_data_set_cmpl_evt_param (C++ *struct*), 264
esp_ble_gap_cb_param_t::ble_ext_adv_data_set_cmpl_evt_param::status (C++ *member*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_report_param (C++ *struct*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_report_param::params (C++ *member*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_scan_rsp_set_cmpl_evt_param (C++ *struct*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_scan_rsp_set_cmpl_evt_param::status (C++ *member*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_set_rand_addr_cmpl_evt_param (C++ *struct*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_set_rand_addr_cmpl_evt_param::status (C++ *member*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_set_remove_cmpl_evt_param (C++ *struct*), 265
esp_ble_gap_cb_param_t::ble_ext_adv_set_remove_cmpl_evt_param::status (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_adv_start_cmpl_evt_param (C++ *struct*), 266
esp_ble_gap_cb_param_t::ble_ext_adv_start_cmpl_evt_param::status (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_adv_stop_cmpl_evt_param (C++ *struct*), 266
esp_ble_gap_cb_param_t::ble_ext_adv_stop_cmpl_evt_param::adv_instance (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_adv_stop_cmpl_evt_param::completed_event (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_conn_params_set_cmpl_param (C++ *struct*), 266
esp_ble_gap_cb_param_t::ble_ext_conn_params_set_cmpl_param::conn_idx (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_conn_params_set_cmpl_param::status (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_scan_start_cmpl_param (C++ *struct*), 266
esp_ble_gap_cb_param_t::ble_ext_scan_start_cmpl_param::channel_selection_algorithm (C++ *member*), 266
esp_ble_gap_cb_param_t::ble_ext_scan_stop_cmpl_param (C++ *struct*), 266
esp_ble_gap_cb_param_t::ble_ext_scan_stop_cmpl_param (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param (C++ *struct*), 267
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::num_of_bond_dev (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::status (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_get_dev_name_cmpl_evt_param (C++ *struct*), 267
esp_ble_gap_cb_param_t::ble_get_dev_name_cmpl_evt_param (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_get_dev_name_cmpl_evt_param::status (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_param (C++ *struct*), 267
esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_param (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_period_adv_add_dev_cmpl_evt_param (C++ *struct*), 267
esp_ble_gap_cb_param_t::ble_period_adv_add_dev_cmpl_evt_param::status (C++ *member*), 267
esp_ble_gap_cb_param_t::ble_period_adv_clear_dev_cmpl_evt_param (C++ *struct*), 267
esp_ble_gap_cb_param_t::ble_period_adv_clear_dev_cmpl_evt_param::status (C++ *member*), 268
esp_ble_gap_cb_param_t::ble_period_adv_create_sync_cmpl_evt_param (C++ *struct*), 268
esp_ble_gap_cb_param_t::ble_period_adv_create_sync_cmpl_evt_param::status (C++ *member*), 268
esp_ble_gap_cb_param_t::ble_period_adv_remove_dev_cmpl_evt_param (C++ *struct*), 268
esp_ble_gap_cb_param_t::ble_period_adv_remove_dev_cmpl_evt_param::status (C++ *member*), 268
esp_ble_gap_cb_param_t::ble_period_adv_sync_cancel_cmpl_evt_param (C++ *struct*), 268
esp_ble_gap_cb_param_t::ble_period_adv_sync_cancel_cmpl_evt_param::status

(C++ member), 268
esp_ble_gap_cb_param_t::ble_period_adv_sync_terminate_cmpl_param_t::adv_handle (C++ struct), 271
(C++ member), 268
esp_ble_gap_cb_param_t::ble_period_adv_sync_terminate_cmpl_param_t::status (C++ member), 271
(C++ member), 268
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_cmpl_param_t (C++ struct), 271
(C++ struct), 268
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_cmpl_param_t::status (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_recv_enable_cmpl_param_t (C++ struct), 271
(C++ struct), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_recv_enable_cmpl_param_t::status (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_transfer_cmpl_param_t (C++ member), 271
(C++ struct), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_transfer_cmpl_param_t::status (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_transfer_cmpl_param_t (C++ member), 271
(C++ struct), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_transfer_cmpl_param_t::conn_handle (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_transfer_cmpl_param_t::periodic_adv_sync_tra
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_info_transfer_cmpl_param_t::status (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_pams_cmpl_param_t (C++ member), 271
(C++ struct), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_pams_cmpl_param_t::conn_handle (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_periodic_adv_set_pams_cmpl_param_t::status (C++ member), 271
(C++ member), 269
esp_ble_gap_cb_param_t::ble_phy_update_cmpl_param (C++ struct), 272
(C++ struct), 269
esp_ble_gap_cb_param_t::ble_phy_update_cmpl_param::status (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_phy_update_cmpl_param_t (C++ member), 272
(C++ struct), 270
esp_ble_gap_cb_param_t::ble_phy_update_cmpl_param_t::status (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_phy_update_cmpl_param_t::tx_phy (C++ member), 272
(C++ struct), 270
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_param_t (C++ struct), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_param_t::max_rx_time (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_param_t::max_rx_time (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_param_t::max_tx_time (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_param_t::max_tx_time (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_phy_cmpl_evt_param (C++ struct), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_phy_cmpl_evt_param::conn_handle (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_phy_cmpl_evt_param::rx_phy (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_phy_cmpl_evt_param::status (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_phy_cmpl_evt_param::tx_phy (C++ member), 272
(C++ member), 270
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param_t (C++ struct), 273
(C++ struct), 270
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param_t::conn_handle (C++ member), 273
(C++ member), 271
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param_t::rssi (C++ member), 273
(C++ member), 271
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param_t::status (C++ member), 273
(C++ member), 271
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_param_t (C++ member), 273
(C++ struct), 271
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_param_t::status (C++ member), 273
(C++ member), 271

(C++ member), 273
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param::status (C++ member), 273
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param (C++ struct), 273
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::bd_addr (C++ member), 273
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::status (C++ member), 273
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param (C++ struct), 273
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param::status (C++ member), 273
esp_ble_gap_cb_param_t::ble_scan_req_received_evt_param (C++ struct), 273
esp_ble_gap_cb_param_t::ble_scan_req_received_evt_param::adv_addr (C++ member), 273
esp_ble_gap_cb_param_t::ble_scan_req_received_evt_param::scan_addr_type (C++ member), 273
esp_ble_gap_cb_param_t::ble_scan_result_evt_param (C++ struct), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::adv_data_len (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::adv_data (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_addr_type (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_evt_type (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::dev_type (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::flag (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::num_resps (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::rssi (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::scan_rsp_len (C++ member), 274
esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param (C++ struct), 274
esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::status (C++ member), 275
esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param (C++ struct), 275
esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param::status (C++ member), 275
esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param (C++ struct), 275
esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param::status (C++ member), 275
esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param (C++ struct), 275
esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param::status (C++ member), 275
esp_ble_gap_cb_param_t::ble_security (C++ member), 259
esp_ble_gap_cb_param_t::ble_set_channels_evt_param (C++ member), 260
esp_ble_gap_cb_param_t::ble_set_channels_evt_param::status (C++ struct), 275
esp_ble_gap_cb_param_t::ble_set_ext_scan_params_cmpl_evt_param (C++ member), 275
esp_ble_gap_cb_param_t::ble_set_ext_scan_params_cmpl_evt_param::status (C++ struct), 275
esp_ble_gap_cb_param_t::ble_set_past_params_cmpl_evt_param (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_past_params_cmpl_evt_param::addr_type (C++ struct), 276
esp_ble_gap_cb_param_t::ble_set_past_params_cmpl_evt_param::bd_addr (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_past_params_cmpl_evt_param::status (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_perf_def_phy_cmpl_evt_param (C++ struct), 276
esp_ble_gap_cb_param_t::ble_set_perf_def_phy_cmpl_evt_param::status (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_perf_phy_cmpl_evt_param (C++ struct), 276
esp_ble_gap_cb_param_t::ble_set_perf_phy_cmpl_evt_param::addr_type (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_perf_phy_cmpl_evt_param::bd_addr (C++ member), 276
esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param (C++ struct), 276
esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::status (C++ member), 276
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param (C++ struct), 276
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::conn_int (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::latency (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::status (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::supervision_tout (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::update_params (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param (C++ struct), 277
esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param::status (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param::subcode (C++ member), 277
esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param::type (C++ member), 277

(*C++ member*), 277
esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param (*C++ member*), 277
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param (*C++ struct*), 277
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param::operation (*C++ member*), 277
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param::status (*C++ member*), 278
esp_ble_gap_cb_param_t::channel_sel_alg (*C++ member*), 262
esp_ble_gap_cb_param_t::clear_bond_dev_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::dtm_state_update (*C++ member*), 262
esp_ble_gap_cb_param_t::ext_adv_clear (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_data_set (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_remove (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_report (*C++ member*), 262
esp_ble_gap_cb_param_t::ext_adv_set_params (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_set_rand_addr (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_start (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_adv_stop (*C++ member*), 260
esp_ble_gap_cb_param_t::ext_conn_params_set (*C++ member*), 261
esp_ble_gap_cb_param_t::ext_scan_start (*C++ member*), 261
esp_ble_gap_cb_param_t::ext_scan_stop (*C++ member*), 261
esp_ble_gap_cb_param_t::get_bond_dev_cmpl (*C++ member*), 260
esp_ble_gap_cb_param_t::get_dev_name_cmpl (*C++ member*), 258
esp_ble_gap_cb_param_t::local_privacy_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::past_received (*C++ member*), 262
esp_ble_gap_cb_param_t::period_adv_add_dev (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_clear_dev (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_create_sync (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_data_set (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_recv_enable (*C++ member*), 262
esp_ble_gap_cb_param_t::period_adv_remove_dev (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_report (*C++ member*), 262
esp_ble_gap_cb_param_t::period_adv_sync_establish_cmpl (*C++ member*), 262
esp_ble_gap_cb_param_t::period_adv_sync_lost_cmpl (*C++ member*), 262
esp_ble_gap_cb_param_t::period_adv_sync_params (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_start (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_stop (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_sync_cancel (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_sync_term (*C++ member*), 261
esp_ble_gap_cb_param_t::period_adv_sync_trans (*C++ member*), 262
esp_ble_gap_cb_param_t::periodic_adv_sync_estab (*C++ member*), 262
esp_ble_gap_cb_param_t::periodic_adv_sync_lost (*C++ member*), 262
esp_ble_gap_cb_param_t::peroid_adv_set_params (*C++ member*), 261
esp_ble_gap_cb_param_t::phy_update (*C++ member*), 262
esp_ble_gap_cb_param_t::pkt_data_length_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::read_phy (*C++ member*), 260
esp_ble_gap_cb_param_t::read_rssi_cmpl (*C++ member*), 260
esp_ble_gap_cb_param_t::remove_bond_dev_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::scan_param_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::scan_req_received (*C++ member*), 261
esp_ble_gap_cb_param_t::scan_rsp_data_cmpl (*C++ member*), 258
esp_ble_gap_cb_param_t::scan_rsp_data_raw_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::scan_rsp_set (*C++ member*), 260
esp_ble_gap_cb_param_t::scan_rst (*C++ member*), 259
esp_ble_gap_cb_param_t::scan_start_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::scan_stop_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::set_ext_scan_params (*C++ member*), 261
esp_ble_gap_cb_param_t::set_past_params (*C++ member*), 262
esp_ble_gap_cb_param_t::set_perf_def_phy (*C++ member*), 260
esp_ble_gap_cb_param_t::set_perf_phy (*C++ member*), 260
esp_ble_gap_cb_param_t::set_rand_addr_cmpl (*C++ member*), 259
esp_ble_gap_cb_param_t::update_conn_params (*C++ member*), 259
esp_ble_gap_cb_param_t::update_duplicate_exceptio

(C++ *member*), 260
esp_ble_gap_cb_param_t::update_whitelist_cmpl (C++ *member*), 260
esp_ble_gap_cb_param_t::vendor_cmd_cmpl (C++ *member*), 262
esp_ble_gap_cb_param_t::vendor_cmd_cmpl_evt (C++ *struct*), 278
esp_ble_gap_cb_param_t::vendor_cmd_cmpl_evt_param::opcode (C++ *member*), 278
esp_ble_gap_cb_param_t::vendor_cmd_cmpl_evt_param::param_len (C++ *member*), 278
esp_ble_gap_cb_param_t::vendor_cmd_cmpl_evt_param::status (C++ *member*), 278
esp_ble_gap_clean_duplicate_scan_exception_list (C++ *function*), 249
esp_ble_gap_clear_advertising (C++ *function*), 257
esp_ble_gap_clear_rand_addr (C++ *function*), 246
esp_ble_gap_clear_whitelist (C++ *function*), 247
esp_ble_gap_config_adv_data (C++ *function*), 245
esp_ble_gap_config_adv_data_raw (C++ *function*), 248
esp_ble_gap_config_ext_adv_data_raw (C++ *function*), 253
esp_ble_gap_config_ext_scan_rsp_data_raw (C++ *function*), 253
esp_ble_gap_config_local_icon (C++ *function*), 247
esp_ble_gap_config_local_privacy (C++ *function*), 247
esp_ble_gap_config_periodic_adv_data_raw (C++ *function*), 254
esp_ble_gap_config_scan_rsp_data_raw (C++ *function*), 248
esp_ble_gap_conn_params_t (C++ *struct*), 288
esp_ble_gap_conn_params_t::interval_max (C++ *member*), 288
esp_ble_gap_conn_params_t::interval_min (C++ *member*), 288
esp_ble_gap_conn_params_t::latency (C++ *member*), 288
esp_ble_gap_conn_params_t::max_ce_len (C++ *member*), 289
esp_ble_gap_conn_params_t::min_ce_len (C++ *member*), 289
esp_ble_gap_conn_params_t::scan_interval (C++ *member*), 288
esp_ble_gap_conn_params_t::scan_window (C++ *member*), 288
esp_ble_gap_conn_params_t::supervision_timeout (C++ *member*), 289
esp_ble_gap_disconnect (C++ *function*), 251
ESP_BLE_GAP_EXT_ADV_DATA_COMPLETE (C *macro*), 301
ESP_BLE_GAP_EXT_ADV_DATA_INCOMPLETE (C *macro*), 302
esp_ble_gap_ext_adv_data_status_t (C++ *type*), 303
ESP_BLE_GAP_EXT_ADV_DATA_TRUNCATED (C *macro*), 302
esp_ble_gap_ext_adv_params_t (C++ *struct*), 286
esp_ble_gap_ext_adv_params_t::channel_map (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::filter_policy (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::interval_max (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::interval_min (C++ *member*), 286
esp_ble_gap_ext_adv_params_t::max_skip (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::own_addr_type (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::peer_addr (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::peer_addr_type (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::primary_phy (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::scan_req_notif (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::secondary_phy (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::sid (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::tx_power (C++ *member*), 287
esp_ble_gap_ext_adv_params_t::type (C++ *member*), 286
esp_ble_gap_ext_adv_report_t (C++ *struct*), 290
esp_ble_gap_ext_adv_report_t::addr (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::addr_type (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::adv_data (C++ *member*), 291
esp_ble_gap_ext_adv_report_t::adv_data_len (C++ *member*), 291
esp_ble_gap_ext_adv_report_t::data_status (C++ *member*), 291
esp_ble_gap_ext_adv_report_t::dir_addr (C++ *member*), 291
esp_ble_gap_ext_adv_report_t::dir_addr_type (C++ *member*), 291
esp_ble_gap_ext_adv_report_t::event_type (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::per_adv_interval (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::primary_phy (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::rssi

(C++ *member*), 290
esp_ble_gap_ext_adv_report_t::secondly_phy (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::sid (C++ *member*), 290
esp_ble_gap_ext_adv_report_t::tx_power (C++ *member*), 290
esp_ble_gap_ext_adv_set_clear (C++ *function*), 254
esp_ble_gap_ext_adv_set_params (C++ *function*), 253
esp_ble_gap_ext_adv_set_rand_addr (C++ *function*), 252
esp_ble_gap_ext_adv_set_remove (C++ *function*), 253
esp_ble_gap_ext_adv_start (C++ *function*), 253
esp_ble_gap_ext_adv_stop (C++ *function*), 253
esp_ble_gap_ext_adv_t (C++ *struct*), 289
esp_ble_gap_ext_adv_t::duration (C++ *member*), 289
esp_ble_gap_ext_adv_t::instance (C++ *member*), 289
esp_ble_gap_ext_adv_t::max_events (C++ *member*), 289
ESP_BLE_GAP_EXT_SCAN_CFG_CODE_MASK (C *macro*), 301
ESP_BLE_GAP_EXT_SCAN_CFG_UNCODE_MASK (C *macro*), 301
esp_ble_gap_get_callback (C++ *function*), 245
esp_ble_gap_get_device_name (C++ *function*), 248
esp_ble_gap_get_local_used_addr (C++ *function*), 248
esp_ble_gap_get_whitelist_size (C++ *function*), 247
ESP_BLE_GAP_NO_PREFER_RECEIVE_PHY (C *macro*), 301
ESP_BLE_GAP_NO_PREFER_TRANSMIT_PHY (C *macro*), 301
ESP_BLE_GAP_PAST_MODE_DUP_FILTER_DISABLED (C *macro*), 303
ESP_BLE_GAP_PAST_MODE_DUP_FILTER_ENABLED (C *macro*), 303
ESP_BLE_GAP_PAST_MODE_NO_REPORT_EVT (C *macro*), 303
ESP_BLE_GAP_PAST_MODE_NO_SYNC_EVT (C *macro*), 302
esp_ble_gap_past_mode_t (C++ *type*), 304
esp_ble_gap_past_params_t (C++ *struct*), 293
esp_ble_gap_past_params_t::cte_type (C++ *member*), 293
esp_ble_gap_past_params_t::mode (C++ *member*), 293
esp_ble_gap_past_params_t::skip (C++ *member*), 293
esp_ble_gap_past_params_t::sync_timeout (C++ *member*), 293
esp_ble_gap_periodic_adv_add_dev_to_list (C++ *function*), 255
esp_ble_gap_periodic_adv_clear_dev (C++ *function*), 255
esp_ble_gap_periodic_adv_create_sync (C++ *function*), 255
esp_ble_gap_periodic_adv_params_t (C++ *struct*), 289
esp_ble_gap_periodic_adv_params_t::interval_max (C++ *member*), 289
esp_ble_gap_periodic_adv_params_t::interval_min (C++ *member*), 289
esp_ble_gap_periodic_adv_params_t::properties (C++ *member*), 289
esp_ble_gap_periodic_adv_recv_enable (C++ *function*), 256
esp_ble_gap_periodic_adv_remove_dev_from_list (C++ *function*), 255
esp_ble_gap_periodic_adv_report_t (C++ *struct*), 291
esp_ble_gap_periodic_adv_report_t::data (C++ *member*), 291
esp_ble_gap_periodic_adv_report_t::data_length (C++ *member*), 291
esp_ble_gap_periodic_adv_report_t::data_status (C++ *member*), 291
esp_ble_gap_periodic_adv_report_t::rssi (C++ *member*), 291
esp_ble_gap_periodic_adv_report_t::sync_handle (C++ *member*), 291
esp_ble_gap_periodic_adv_report_t::tx_power (C++ *member*), 291
esp_ble_gap_periodic_adv_set_info_trans (C++ *function*), 256
esp_ble_gap_periodic_adv_set_params (C++ *function*), 254
esp_ble_gap_periodic_adv_start (C++ *function*), 254
esp_ble_gap_periodic_adv_stop (C++ *function*), 254
esp_ble_gap_periodic_adv_sync_cancel (C++ *function*), 255
esp_ble_gap_periodic_adv_sync_estab_t (C++ *struct*), 291
esp_ble_gap_periodic_adv_sync_estab_t::addr_type (C++ *member*), 292
esp_ble_gap_periodic_adv_sync_estab_t::adv_addr (C++ *member*), 292
esp_ble_gap_periodic_adv_sync_estab_t::adv_clk_ac (C++ *member*), 292
esp_ble_gap_periodic_adv_sync_estab_t::adv_phy (C++ *member*), 292
esp_ble_gap_periodic_adv_sync_estab_t::period_adv (C++ *member*), 292
esp_ble_gap_periodic_adv_sync_estab_t::sid (C++ *member*), 292

esp_ble_gap_periodic_adv_sync_estab_t::status (C++ member), 291
esp_ble_gap_periodic_adv_sync_estab_t::sync_handle (C++ member), 292
esp_ble_gap_periodic_adv_sync_params_t (C++ struct), 289
esp_ble_gap_periodic_adv_sync_params_t::adv_addr (C++ member), 290
esp_ble_gap_periodic_adv_sync_params_t::adv_addr_type (C++ member), 290
esp_ble_gap_periodic_adv_sync_params_t::filter_policy (C++ member), 289
esp_ble_gap_periodic_adv_sync_params_t::sid (C++ member), 290
esp_ble_gap_periodic_adv_sync_params_t::skip (C++ member), 290
esp_ble_gap_periodic_adv_sync_params_t::sync_timeout (C++ member), 290
esp_ble_gap_periodic_adv_sync_terminate (C++ function), 255
esp_ble_gap_periodic_adv_sync_trans (C++ function), 256
ESP_BLE_GAP_PHY_1M (C macro), 300
ESP_BLE_GAP_PHY_1M_PREF_MASK (C macro), 301
ESP_BLE_GAP_PHY_2M (C macro), 301
ESP_BLE_GAP_PHY_2M_PREF_MASK (C macro), 301
ESP_BLE_GAP_PHY_CODED (C macro), 301
ESP_BLE_GAP_PHY_CODED_PREF_MASK (C macro), 301
esp_ble_gap_phy_mask_t (C++ type), 303
ESP_BLE_GAP_PHY_OPTIONS_NO_PREF (C macro), 301
ESP_BLE_GAP_PHY_OPTIONS_PREF_S2_CODING (C macro), 301
ESP_BLE_GAP_PHY_OPTIONS_PREF_S8_CODING (C macro), 301
esp_ble_gap_phy_t (C++ type), 303
esp_ble_gap_prefer_ext_connect_params_set (C++ function), 255
esp_ble_gap_prefer_phy_options_t (C++ type), 303
ESP_BLE_GAP_PRI_PHY_1M (C macro), 301
ESP_BLE_GAP_PRI_PHY_CODED (C macro), 301
esp_ble_gap_pri_phy_t (C++ type), 303
esp_ble_gap_read_phy (C++ function), 252
esp_ble_gap_read_rssi (C++ function), 248
esp_ble_gap_register_callback (C++ function), 245
esp_ble_gap_remove_duplicate_scan_exceptional_device_list (C++ function), 249
esp_ble_gap_security_rsp (C++ function), 250
esp_ble_gap_set_device_name (C++ function), 248
ESP_BLE_GAP_SET_EXT_ADV_PROP_ANON_ADV (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_CONNECTABLE (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_DIRECTED (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_HD_DIRECTED (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_INCLUDE_TX_PWR (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_HD_DIR (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_IND (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_LD_DIR (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_NONCONN (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_LEGACY_SCAN (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_MASK (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_NONCONN_NONSCANNABLE (C macro), 300
ESP_BLE_GAP_SET_EXT_ADV_PROP_SCANNABLE (C macro), 300
esp_ble_gap_set_ext_scan_params (C++ function), 254
esp_ble_gap_set_periodic_adv_sync_trans_params (C++ function), 256
esp_ble_gap_set_pkt_data_len (C++ function), 246
esp_ble_gap_set_prefer_conn_params (C++ function), 247
esp_ble_gap_set_preferred_default_phy (C++ function), 252
esp_ble_gap_set_preferred_phy (C++ function), 252
esp_ble_gap_set_rand_addr (C++ function), 246
esp_ble_gap_set_scan_params (C++ function), 245
esp_ble_gap_set_security_param (C++ function), 249
esp_ble_gap_start_advertising (C++ function), 246
esp_ble_gap_start_ext_scan (C++ function), 254
esp_ble_gap_start_scanning (C++ function), 245
esp_ble_gap_stop_advertising (C++ function), 246
esp_ble_gap_stop_ext_scan (C++ function), 254
esp_ble_gap_stop_scanning (C++ function), 246
ESP_BLE_GAP_SYNC_POLICY_BY_ADV_INFO (C macro), 302

ESP_BLE_GAP_SYNC_POLICY_BY_PERIODIC_LIST (*C macro*), 302
esp_ble_gap_sync_t (*C++ type*), 303
esp_ble_gap_update_conn_params (*C++ function*), 246
esp_ble_gap_update_whitelist (*C++ function*), 247
esp_ble_gap_vendor_command_send (*C++ function*), 257
esp_ble_gattc_app_register (*C++ function*), 347
esp_ble_gattc_app_unregister (*C++ function*), 347
esp_ble_gattc_aux_open (*C++ function*), 348
esp_ble_gattc_cache_assoc (*C++ function*), 354
esp_ble_gattc_cache_clean (*C++ function*), 355
esp_ble_gattc_cache_get_addr_list (*C++ function*), 355
esp_ble_gattc_cache_refresh (*C++ function*), 354
esp_ble_gattc_cb_param_t (*C++ union*), 355
esp_ble_gattc_cb_param_t::cfg_mtu (*C++ member*), 355
esp_ble_gattc_cb_param_t::close (*C++ member*), 355
esp_ble_gattc_cb_param_t::congest (*C++ member*), 356
esp_ble_gattc_cb_param_t::connect (*C++ member*), 356
esp_ble_gattc_cb_param_t::dis_srvc_cmpl (*C++ member*), 356
esp_ble_gattc_cb_param_t::disconnect (*C++ member*), 356
esp_ble_gattc_cb_param_t::exec_cmpl (*C++ member*), 356
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param (*C++ struct*), 356
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::conn_id (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::mtu (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::status (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_close_evt_param (*C++ struct*), 357
esp_ble_gattc_cb_param_t::gattc_close_evt_param::conn_id (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_close_evt_param::reason (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_close_evt_param::remote_bda (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_close_evt_param::status (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_congest_evt_param (*C++ struct*), 357
esp_ble_gattc_cb_param_t::gattc_congest_evt_param::conn_id (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_connect_evt_param (*C++ struct*), 357
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::conn_id (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::link_role (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::remote_bda (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::remote_bda_type (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::transport (*C++ member*), 357
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::tx_mtu (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_dis_srvc_cmpl_evt_param (*C++ struct*), 358
esp_ble_gattc_cb_param_t::gattc_dis_srvc_cmpl_evt_param::conn_id (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_dis_srvc_cmpl_evt_param::status (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param (*C++ struct*), 358
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::conn_id (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::reason (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::remote_bda (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param (*C++ struct*), 358
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::status (*C++ member*), 358
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::count (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::is_notify (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::remote_bda (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param::handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param::value (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_descr_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param::num_handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param::value (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_read_multi_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::searched_service_source (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::end_handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::is_primary (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::start_handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::srvc_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param::handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_descr_evt_param::value_type (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param (*C++ struct*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param::conn_id (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param::handle (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param::status (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param::value_len (*C++ member*), 359
esp_ble_gattc_cb_param_t::gattc_write_char_evt_param::value_type (*C++ member*), 359

(C++ member), 359
esp_ble_gattc_cb_param_t::gattc_open_evt_param (C++ struct), 359
esp_ble_gattc_cb_param_t::gattc_open_evt_param::addr (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_open_evt_param::conn_id (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_open_evt_param::mtu (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_open_evt_param::remote_bda (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_open_evt_param::status (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param (C++ struct), 360
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::conn_id (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::is_full (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::status (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param (C++ struct), 360
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::conn_id (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::handle (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::status (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value (C++ member), 360
esp_ble_gattc_cb_param_t::gattc_reg_evt_param (C++ struct), 360
esp_ble_gattc_cb_param_t::gattc_reg_evt_param::app_id (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_reg_evt_param::gattc_if (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param (C++ struct), 361
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::conn_id (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::handle (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::status (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param (C++ struct), 361
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::conn_id (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::searched_service_source (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::status (C++ member), 361
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param (C++ struct), 361
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::conn_id (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::end_handle (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::is_primary (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::start_handle (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::srvc_id (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp_evt_param (C++ struct), 362
esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp_evt_param::addr (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp_evt_param::status (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param (C++ struct), 362
esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param::remote_bda (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param (C++ struct), 362
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param::conn_id (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param::handle (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param::status (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_write_evt_param (C++ struct), 362
esp_ble_gattc_cb_param_t::gattc_write_evt_param::conn_id (C++ member), 362
esp_ble_gattc_cb_param_t::gattc_write_evt_param::handle (C++ member), 363
esp_ble_gattc_cb_param_t::gattc_write_evt_param::status (C++ member), 363
esp_ble_gattc_cb_param_t::gattc_write_evt_param::value (C++ member), 362
esp_ble_gattc_cb_param_t::get_addr_list (C++ member), 356
esp_ble_gattc_cb_param_t::notify (C++ member), 356
esp_ble_gattc_cb_param_t::open (C++ member), 355
esp_ble_gattc_cb_param_t::queue_full (C++ member), 356
esp_ble_gattc_cb_param_t::read (C++ member), 356
esp_ble_gattc_cb_param_t::reg (C++ member), 355
esp_ble_gattc_cb_param_t::reg_for_notify (C++ member), 356
esp_ble_gattc_cb_param_t::search_cmpl (C++ member), 355
esp_ble_gattc_cb_param_t::search_res (C++ member), 356
esp_ble_gattc_cb_param_t::srvc_chg (C++ member), 356
esp_ble_gattc_cb_param_t::unreg_for_notify (C++ member), 356
esp_ble_gattc_cb_param_t::write (C++ member), 356
esp_ble_gattc_close (C++ function), 348
esp_ble_gattc_execute_write (C++ function), 354

esp_ble_gattc_get_all_char (C++ function), 349
esp_ble_gattc_get_all_descr (C++ function), 349
esp_ble_gattc_get_attr_count (C++ function), 351
esp_ble_gattc_get_callback (C++ function), 347
esp_ble_gattc_get_char_by_uuid (C++ function), 349
esp_ble_gattc_get_db (C++ function), 351
esp_ble_gattc_get_descr_by_char_handle (C++ function), 350
esp_ble_gattc_get_descr_by_uuid (C++ function), 350
esp_ble_gattc_get_include_service (C++ function), 350
esp_ble_gattc_get_service (C++ function), 349
esp_ble_gattc_open (C++ function), 348
esp_ble_gattc_prepare_write (C++ function), 353
esp_ble_gattc_prepare_write_char_descr (C++ function), 353
esp_ble_gattc_read_by_type (C++ function), 352
esp_ble_gattc_read_char (C++ function), 351
esp_ble_gattc_read_char_descr (C++ function), 352
esp_ble_gattc_read_multiple (C++ function), 352
esp_ble_gattc_read_multiple_variable (C++ function), 352
esp_ble_gattc_register_callback (C++ function), 347
esp_ble_gattc_register_for_notify (C++ function), 354
esp_ble_gattc_search_service (C++ function), 348
esp_ble_gattc_send_mtu_req (C++ function), 348
esp_ble_gattc_unregister_for_notify (C++ function), 354
esp_ble_gattc_write_char (C++ function), 353
esp_ble_gattc_write_char_descr (C++ function), 353
esp_ble_gatts_add_char (C++ function), 333
esp_ble_gatts_add_char_descr (C++ function), 333
esp_ble_gatts_add_included_service (C++ function), 333
esp_ble_gatts_app_register (C++ function), 332
esp_ble_gatts_app_unregister (C++ function), 332
esp_ble_gatts_cb_param_t (C++ union), 336
esp_ble_gatts_cb_param_t::add_attr_tab (C++ member), 337
esp_ble_gatts_cb_param_t::add_char (C++ member), 336
esp_ble_gatts_cb_param_t::add_char_descr (C++ member), 336
esp_ble_gatts_cb_param_t::add_incl_srvc (C++ member), 336
esp_ble_gatts_cb_param_t::cancel_open (C++ member), 337
esp_ble_gatts_cb_param_t::close (C++ member), 337
esp_ble_gatts_cb_param_t::conf (C++ member), 336
esp_ble_gatts_cb_param_t::congest (C++ member), 337
esp_ble_gatts_cb_param_t::connect (C++ member), 337
esp_ble_gatts_cb_param_t::create (C++ member), 336
esp_ble_gatts_cb_param_t::del (C++ member), 336
esp_ble_gatts_cb_param_t::disconnect (C++ member), 337
esp_ble_gatts_cb_param_t::exec_write (C++ member), 336
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ struct), 337
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ member), 337
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ member), 337
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ member), 337
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_ (C++ member), 337
esp_ble_gatts_cb_param_t::gatts_add_char_descr_ev (C++ struct), 338
esp_ble_gatts_cb_param_t::gatts_add_char_descr_ev (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_descr_ev (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_descr_ev (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_descr_ev (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_evt_para (C++ struct), 338
esp_ble_gatts_cb_param_t::gatts_add_char_evt_para (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_evt_para (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_evt_para (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_char_evt_para (C++ member), 338
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt

(C++ *struct*), 338
esp_ble_gatts_cb_param_t::gatts_add_inc (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_add_inc (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_add_inc (C++ *member*), 338
esp_ble_gatts_cb_param_t::gatts_cancel (C++ *struct*), 339
esp_ble_gatts_cb_param_t::gatts_cancel (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_close_e (C++ *struct*), 339
esp_ble_gatts_cb_param_t::gatts_close_e (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_close_e (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *struct*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_conf_ev (C++ *member*), 339
esp_ble_gatts_cb_param_t::gatts_congest (C++ *struct*), 340
esp_ble_gatts_cb_param_t::gatts_congest (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_congest (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *struct*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_connect (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_create_ (C++ *struct*), 340
esp_ble_gatts_cb_param_t::gatts_create_ (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_create_ (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_create_ (C++ *member*), 340
esp_ble_gatts_cb_param_t::gatts_delete_ (C++ *struct*), 341
esp_ble_gatts_cb_param_t::gatts_delete_ (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_delete_ (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_disconn (C++ *struct*), 341
esp_ble_gatts_cb_param_t::gatts_disconn (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_disconn (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_disconn (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_exec_wr (C++ *struct*), 341
esp_ble_gatts_cb_param_t::gatts_exec_wr (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_exec_wr (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_exec_wr (C++ *member*), 341
esp_ble_gatts_cb_param_t::gatts_mtu_evt_ (C++ *struct*), 342
esp_ble_gatts_cb_param_t::gatts_mtu_evt_ (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_mtu_evt_ (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_open_evt (C++ *struct*), 342
esp_ble_gatts_cb_param_t::gatts_open_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *struct*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_read_evt (C++ *member*), 342
esp_ble_gatts_cb_param_t::gatts_reg_evt (C++ *struct*), 343
esp_ble_gatts_cb_param_t::gatts_reg_evt (C++ *member*), 343
esp_ble_gatts_cb_param_t::gatts_reg_evt (C++ *member*), 343
esp_ble_gatts_cb_param_t::gatts_rsp_evt (C++ *struct*), 343
esp_ble_gatts_cb_param_t::gatts_rsp_evt

(*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::status (*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_send_service_change_evt_param (*C++ struct*), 343
esp_ble_gatts_cb_param_t::gatts_send_service_change_evt_param::status (*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param (*C++ struct*), 343
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::set_attr_val (*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::srvc_handle (*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::status (*C++ member*), 343
esp_ble_gatts_cb_param_t::gatts_start_evt_param (*C++ struct*), 343
esp_ble_gatts_cb_param_t::gatts_start_evt_param::service_handle (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_start_evt_param::status (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_stop_evt_param (*C++ struct*), 344
esp_ble_gatts_cb_param_t::gatts_stop_evt_param::service_handle (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_stop_evt_param::status (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param (*C++ struct*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::bda (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::conn_id (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::handle (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::is_prep (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::len (*C++ member*), 345
esp_ble_gatts_cb_param_t::gatts_write_evt_param::need_rsp (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::offset (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::trans_id (*C++ member*), 344
esp_ble_gatts_cb_param_t::gatts_write_evt_param::value (*C++ member*), 345
esp_ble_gatts_cb_param_t::mtu (*C++ member*), 336
esp_ble_gatts_cb_param_t::open (*C++ member*), 337
esp_ble_gatts_cb_param_t::read (*C++ member*), 336
esp_ble_gatts_cb_param_t::reg (*C++ member*), 336
esp_ble_gatts_cb_param_t::rsp (*C++ member*), 337
esp_ble_gatts_cb_param_t::service_change (*C++ member*), 337
esp_ble_gatts_cb_param_t::set_attr_val (*C++ member*), 337
esp_ble_gatts_cb_param_t::start (*C++ member*), 336
esp_ble_gatts_cb_param_t::stop (*C++ member*), 336
esp_ble_gatts_cb_param_t::write (*C++ member*), 336
esp_ble_gatts_cb_param_t::set_attr_val (*C++ function*), 335
esp_ble_gatts_create_attr_tab (*C++ function*), 333
esp_ble_gatts_create_service (*C++ function*), 332
esp_ble_gatts_delete_service (*C++ function*), 334
esp_ble_gatts_get_attr_value (*C++ function*), 335
esp_ble_gatts_get_callback (*C++ function*), 332
esp_ble_gatts_open (*C++ function*), 335
esp_ble_gatts_register_callback (*C++ function*), 332
esp_ble_gatts_send_indicate (*C++ function*), 334
esp_ble_gatts_send_response (*C++ function*), 334
esp_ble_gatts_send_service_change_indication (*C++ function*), 335
esp_ble_gatts_set_attr_value (*C++ function*), 335
esp_ble_gatts_show_local_database (*C++ function*), 335
esp_ble_gatts_start_service (*C++ function*), 334
esp_ble_gatts_stop_service (*C++ function*), 334
esp_ble_gatts_bond_device_list (*C++ function*), 251
esp_ble_gatts_bond_device_num (*C++ function*), 250
esp_ble_gatts_offset_conn_params (*C++ function*), 251
ESP_BLE_ID_KEY_MASK (*C macro*), 238
esp_ble_io_cap_t (*C++ type*), 303
ESP_BLE_IS_VALID_PARAM (*C macro*), 237
esp_ble_key_mask_t (*C++ type*), 238
esp_ble_key_t (*C++ struct*), 285
esp_ble_key_t::bd_addr (*C++ member*), 285
esp_ble_key_t::key_type (*C++ member*), 285
esp_ble_key_t::p_key_value (*C++ member*), 285
esp_ble_key_type_t (*C++ type*), 303
esp_ble_key_value_t (*C++ union*), 257
esp_ble_key_value_t::lcsrk_key (*C++ member*), 258
esp_ble_key_value_t::lenc_key (*C++ member*), 258

esp_ble_key_value_t::pcsrk_key (*C++ member*), 258
esp_ble_key_value_t::penc_key (*C++ member*), 257
esp_ble_key_value_t::pid_key (*C++ member*), 258
esp_ble_lcsrk_keys (*C++ struct*), 283
esp_ble_lcsrk_keys::counter (*C++ member*), 284
esp_ble_lcsrk_keys::csrk (*C++ member*), 284
esp_ble_lcsrk_keys::div (*C++ member*), 284
esp_ble_lcsrk_keys::sec_level (*C++ member*), 284
ESP_BLE_LEGACY_ADV_TYPE_DIRECT_IND (*C macro*), 302
ESP_BLE_LEGACY_ADV_TYPE_IND (*C macro*), 302
ESP_BLE_LEGACY_ADV_TYPE_NONCON_IND (*C macro*), 302
ESP_BLE_LEGACY_ADV_TYPE_SCAN_IND (*C macro*), 302
ESP_BLE_LEGACY_ADV_TYPE_SCAN_RSP_TO_ADV_IND (*C macro*), 302
ESP_BLE_LEGACY_ADV_TYPE_SCAN_RSP_TO_ADV_SCAN_IND (*C macro*), 302
esp_ble_lenc_keys_t (*C++ struct*), 283
esp_ble_lenc_keys_t::div (*C++ member*), 283
esp_ble_lenc_keys_t::key_size (*C++ member*), 283
esp_ble_lenc_keys_t::ltk (*C++ member*), 283
esp_ble_lenc_keys_t::sec_level (*C++ member*), 283
ESP_BLE_LINK_KEY_MASK (*C macro*), 238
esp_ble_local_id_keys_t (*C++ struct*), 285
esp_ble_local_id_keys_t::dhk (*C++ member*), 285
esp_ble_local_id_keys_t::ir (*C++ member*), 285
esp_ble_local_id_keys_t::irk (*C++ member*), 285
esp_ble_local_oob_data_t (*C++ struct*), 285
esp_ble_local_oob_data_t::oob_c (*C++ member*), 286
esp_ble_local_oob_data_t::oob_r (*C++ member*), 286
ESP_BLE_MESH_ACTUATOR_BLOCKED_ERROR (*C macro*), 668
ESP_BLE_MESH_ACTUATOR_BLOCKED_WARNING (*C macro*), 668
ESP_BLE_MESH_ADDR_ALL_NODES (*C macro*), 575
ESP_BLE_MESH_ADDR_FRIENDS (*C macro*), 575
ESP_BLE_MESH_ADDR_IS_GROUP (*C macro*), 576
ESP_BLE_MESH_ADDR_IS_RFU (*C macro*), 576
ESP_BLE_MESH_ADDR_IS_UNICAST (*C macro*), 576
ESP_BLE_MESH_ADDR_IS_VIRTUAL (*C macro*), 576
ESP_BLE_MESH_ADDR_PROXIES (*C macro*), 575
ESP_BLE_MESH_ADDR_RELAYS (*C macro*), 575
ESP_BLE_MESH_ADDR_TYPE_PUBLIC (*C macro*), 578
ESP_BLE_MESH_ADDR_TYPE_RANDOM (*C macro*), 578
ESP_BLE_MESH_ADDR_TYPE_RPA_PUBLIC (*C macro*), 578
ESP_BLE_MESH_ADDR_TYPE_RPA_RANDOM (*C macro*), 578
esp_ble_mesh_addr_type_t (*C++ type*), 597
ESP_BLE_MESH_ADDR_UNASSIGNED (*C macro*), 574
ESP_BLE_MESH_ADDR_UNASSIGNED_ELT_ (*C macro*), 575
ESP_BLE_MESH_BATTERY_LOW_ERROR (*C macro*), 667
ESP_BLE_MESH_BATTERY_LOW_WARNING (*C macro*), 667
esp_ble_mesh_bd_addr_t (*C++ type*), 597
ESP_BLE_MESH_BEACON_DISABLED (*C macro*), 575
ESP_BLE_MESH_BEACON_ENABLED (*C macro*), 575
esp_ble_mesh_cb_t (*C++ type*), 596
esp_ble_mesh_cb_type_t (*C++ enum*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_COMPLET (*C++ enumerator*), 599
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_INTPUT_ (*C++ enumerator*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_LINK_CL (*C++ enumerator*), 599
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_LINK_OP (*C++ enumerator*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_OUTPUT_ (*C++ enumerator*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_OUTPUT_ (*C++ enumerator*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_PROV_CB (*C++ enumerator*), 598
esp_ble_mesh_cb_type_t::ESP_BLE_MESH_TYPE_RESET_C (*C++ enumerator*), 599
esp_ble_mesh_cfg_app_key_add_t (*C++ struct*), 638
esp_ble_mesh_cfg_app_key_add_t::app_idx (*C++ member*), 639
esp_ble_mesh_cfg_app_key_add_t::app_key (*C++ member*), 639
esp_ble_mesh_cfg_app_key_add_t::net_idx (*C++ member*), 639
esp_ble_mesh_cfg_app_key_delete_t (*C++ struct*), 644
esp_ble_mesh_cfg_app_key_delete_t::app_idx (*C++ member*), 644
esp_ble_mesh_cfg_app_key_delete_t::net_idx

(C++ *member*), 644
esp_ble_mesh_cfg_app_key_get_t (C++ *struct*), 636
esp_ble_mesh_cfg_app_key_get_t::net_idx (C++ *member*), 636
esp_ble_mesh_cfg_app_key_list_cb_t (C++ *struct*), 651
esp_ble_mesh_cfg_app_key_list_cb_t::app_idx (C++ *member*), 651
esp_ble_mesh_cfg_app_key_list_cb_t::net_idx (C++ *member*), 651
esp_ble_mesh_cfg_app_key_list_cb_t::status (C++ *member*), 651
esp_ble_mesh_cfg_app_key_status_cb_t (C++ *struct*), 648
esp_ble_mesh_cfg_app_key_status_cb_t::app_idx (C++ *member*), 649
esp_ble_mesh_cfg_app_key_status_cb_t::net_idx (C++ *member*), 649
esp_ble_mesh_cfg_app_key_status_cb_t::status (C++ *member*), 648
esp_ble_mesh_cfg_app_key_update_t (C++ *struct*), 644
esp_ble_mesh_cfg_app_key_update_t::app_idx (C++ *member*), 644
esp_ble_mesh_cfg_app_key_update_t::app_key (C++ *member*), 644
esp_ble_mesh_cfg_app_key_update_t::net_idx (C++ *member*), 644
esp_ble_mesh_cfg_beacon_set_t (C++ *struct*), 637
esp_ble_mesh_cfg_beacon_set_t::beacon (C++ *member*), 637
esp_ble_mesh_cfg_beacon_status_cb_t (C++ *struct*), 646
esp_ble_mesh_cfg_beacon_status_cb_t::beacon (C++ *member*), 646
esp_ble_mesh_cfg_client_cb_event_t (C++ *enum*), 658
esp_ble_mesh_cfg_client_cb_event_t::ESP_BLE_MESH_CFG_CLIENT_EVT_MAX (C++ *enumerator*), 658
esp_ble_mesh_cfg_client_cb_event_t::ESP_BLE_MESH_CFG_CLIENT_GET_STATE_EVT (C++ *enumerator*), 658
esp_ble_mesh_cfg_client_cb_event_t::ESP_BLE_MESH_CFG_CLIENT_PUBLISH_EVT (C++ *enumerator*), 658
esp_ble_mesh_cfg_client_cb_event_t::ESP_BLE_MESH_CFG_CLIENT_SET_STATE_EVT (C++ *enumerator*), 658
esp_ble_mesh_cfg_client_cb_event_t::ESP_BLE_MESH_CFG_CLIENT_TIMEOUT_EVT (C++ *enumerator*), 658
esp_ble_mesh_cfg_client_cb_param_t (C++ *struct*), 653
esp_ble_mesh_cfg_client_cb_param_t::error_code (C++ *member*), 653
esp_ble_mesh_cfg_client_cb_param_t::params (C++ *member*), 653
esp_ble_mesh_cfg_client_cb_param_t::status_cb (C++ *member*), 653
esp_ble_mesh_cfg_client_cb_t (C++ *type*), 658
esp_ble_mesh_cfg_client_common_cb_param_t (C++ *union*), 631
esp_ble_mesh_cfg_client_common_cb_param_t::appkey (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::appkey_idx (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::beacon (C++ *member*), 631
esp_ble_mesh_cfg_client_common_cb_param_t::comp_data (C++ *member*), 631
esp_ble_mesh_cfg_client_common_cb_param_t::default_ttl (C++ *member*), 631
esp_ble_mesh_cfg_client_common_cb_param_t::friend (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::gatt_proxy (C++ *member*), 631
esp_ble_mesh_cfg_client_common_cb_param_t::heartbeat_dst (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::heartbeat_pub (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::kr_phase (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::lpn_timeout (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::model_app_idx (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::model_pub (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::model_sub (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::net_trans (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::netkey (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::netkey_idx (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::node_id (C++ *member*), 632
esp_ble_mesh_cfg_client_common_cb_param_t::relay (C++ *member*), 631
esp_ble_mesh_cfg_client_get_state_t (C++ *union*), 628
esp_ble_mesh_cfg_client_get_state_t::app_key_get (C++ *member*), 629
esp_ble_mesh_cfg_client_get_state_t::comp_data_get (C++ *member*), 629
esp_ble_mesh_cfg_client_get_state_t::kr_phase_get (C++ *member*), 629
esp_ble_mesh_cfg_client_get_state_t::lpn_pollto_get (C++ *member*), 629
esp_ble_mesh_cfg_client_get_state_t::model_pub_get (C++ *member*), 629
esp_ble_mesh_cfg_client_get_state_t::node_identity

(C++ member), 629
esp_ble_mesh_cfg_client_get_state_t::sig_model_sub_get (C++ member), 629
esp_ble_mesh_cfg_client_get_state_t::vnd_model_app_get (C++ member), 629
esp_ble_mesh_cfg_client_get_state_t::vnd_model_sub_get (C++ member), 629
esp_ble_mesh_cfg_client_set_state_t (C++ union), 629
esp_ble_mesh_cfg_client_set_state_t::app_key_add (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::app_key_delete (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::app_key_update (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::beacon_set (C++ member), 629
esp_ble_mesh_cfg_client_set_state_t::default_ttl_set (C++ member), 629
esp_ble_mesh_cfg_client_set_state_t::friend_set (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::gatt_proxy_set (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::heartbeat_pub_set (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::heartbeat_sub_set (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::krp_phase_set (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::model_app_bind (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_app_unbind (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::model_pub_set (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_add (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_delete (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_delete_all (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::model_sub_overwrite (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_virtual_addr_add (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_virtual_addr_delete (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::model_sub_virtual_addr_overwrite (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::net_key_add (C++ member), 630
esp_ble_mesh_cfg_client_set_state_t::net_key_delete (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::net_key_update (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::net_transmit_set (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::node_identity_set (C++ member), 631
esp_ble_mesh_cfg_client_set_state_t::relay_set (C++ member), 630
esp_ble_mesh_cfg_comp_data_status_cb_t (C++ struct), 646
esp_ble_mesh_cfg_comp_data_status_cb_t::composition_data (C++ member), 646
esp_ble_mesh_cfg_comp_data_status_cb_t::page (C++ member), 646
esp_ble_mesh_cfg_composition_data_get_t (C++ struct), 635
esp_ble_mesh_cfg_composition_data_get_t::page (C++ member), 635
esp_ble_mesh_cfg_default_ttl_set_t (C++ struct), 637
esp_ble_mesh_cfg_default_ttl_set_t::ttl (C++ member), 638
esp_ble_mesh_cfg_default_ttl_status_cb_t (C++ struct), 646
esp_ble_mesh_cfg_default_ttl_status_cb_t::default_ttl (C++ member), 646
esp_ble_mesh_cfg_friend_set_t (C++ struct), 638
esp_ble_mesh_cfg_friend_set_t::friend_state (C++ member), 638
esp_ble_mesh_cfg_friend_status_cb_t (C++ struct), 649
esp_ble_mesh_cfg_friend_status_cb_t::friend_state (C++ member), 649
esp_ble_mesh_cfg_gatt_proxy_set_t (C++ struct), 638
esp_ble_mesh_cfg_gatt_proxy_set_t::gatt_proxy (C++ member), 638
esp_ble_mesh_cfg_gatt_proxy_status_cb_t (C++ struct), 646
esp_ble_mesh_cfg_gatt_proxy_status_cb_t::gatt_proxy (C++ member), 647
esp_ble_mesh_cfg_hb_pub_status_cb_t (C++ struct), 649
esp_ble_mesh_cfg_hb_pub_status_cb_t::count (C++ member), 650
esp_ble_mesh_cfg_hb_pub_status_cb_t::dst (C++ member), 649
esp_ble_mesh_cfg_hb_pub_status_cb_t::features (C++ member), 650
esp_ble_mesh_cfg_hb_pub_status_cb_t::net_idx (C++ member), 650
esp_ble_mesh_cfg_hb_pub_status_cb_t::period (C++ member), 650
esp_ble_mesh_cfg_hb_pub_status_cb_t::status (C++ member), 649
esp_ble_mesh_cfg_hb_pub_status_cb_t::ttl (C++ member), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t

(C++ *struct*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::count (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::dst (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::max_pkts (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::min_pkts (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::period (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::src (C++ *member*), 650
esp_ble_mesh_cfg_hb_sub_status_cb_t::status (C++ *member*), 650
esp_ble_mesh_cfg_heartbeat_pub_set_t (C++ *struct*), 645
esp_ble_mesh_cfg_heartbeat_pub_set_t::count (C++ *member*), 645
esp_ble_mesh_cfg_heartbeat_pub_set_t::dst (C++ *member*), 645
esp_ble_mesh_cfg_heartbeat_pub_set_t::features (C++ *member*), 645
esp_ble_mesh_cfg_heartbeat_pub_set_t::net_idx (C++ *member*), 646
esp_ble_mesh_cfg_heartbeat_pub_set_t::period (C++ *member*), 645
esp_ble_mesh_cfg_heartbeat_pub_set_t::ttl (C++ *member*), 645
esp_ble_mesh_cfg_heartbeat_sub_set_t (C++ *struct*), 646
esp_ble_mesh_cfg_heartbeat_sub_set_t::dst (C++ *member*), 646
esp_ble_mesh_cfg_heartbeat_sub_set_t::period (C++ *member*), 646
esp_ble_mesh_cfg_heartbeat_sub_set_t::src (C++ *member*), 646
esp_ble_mesh_cfg_kr_phase_get_t (C++ *struct*), 637
esp_ble_mesh_cfg_kr_phase_get_t::net_idx (C++ *member*), 637
esp_ble_mesh_cfg_kr_phase_set_t (C++ *struct*), 645
esp_ble_mesh_cfg_kr_phase_set_t::net_idx (C++ *member*), 645
esp_ble_mesh_cfg_kr_phase_set_t::transition (C++ *member*), 645
esp_ble_mesh_cfg_kr_phase_status_cb_t (C++ *struct*), 652
esp_ble_mesh_cfg_kr_phase_status_cb_t::net_idx (C++ *member*), 652
esp_ble_mesh_cfg_kr_phase_status_cb_t::phase (C++ *member*), 653
esp_ble_mesh_cfg_kr_phase_status_cb_t::status (C++ *member*), 652
esp_ble_mesh_cfg_lpn_polltimeout_get_t (C++ *struct*), 637
esp_ble_mesh_cfg_lpn_polltimeout_get_t::lpn_addr (C++ *member*), 637
esp_ble_mesh_cfg_lpn_pollto_status_cb_t (C++ *struct*), 653
esp_ble_mesh_cfg_lpn_pollto_status_cb_t::lpn_addr (C++ *member*), 653
esp_ble_mesh_cfg_lpn_pollto_status_cb_t::poll_timeout (C++ *member*), 653
esp_ble_mesh_cfg_mod_app_status_cb_t (C++ *struct*), 649
esp_ble_mesh_cfg_mod_app_status_cb_t::app_idx (C++ *member*), 649
esp_ble_mesh_cfg_mod_app_status_cb_t::company_id (C++ *member*), 649
esp_ble_mesh_cfg_mod_app_status_cb_t::element_addr (C++ *member*), 649
esp_ble_mesh_cfg_mod_app_status_cb_t::model_id (C++ *member*), 649
esp_ble_mesh_cfg_mod_app_status_cb_t::status (C++ *member*), 649
esp_ble_mesh_cfg_model_app_bind_t (C++ *struct*), 639
esp_ble_mesh_cfg_model_app_bind_t::company_id (C++ *member*), 639
esp_ble_mesh_cfg_model_app_bind_t::element_addr (C++ *member*), 639
esp_ble_mesh_cfg_model_app_bind_t::model_app_idx (C++ *member*), 639
esp_ble_mesh_cfg_model_app_bind_t::model_id (C++ *member*), 639
esp_ble_mesh_cfg_model_app_list_cb_t (C++ *struct*), 652
esp_ble_mesh_cfg_model_app_list_cb_t::app_idx (C++ *member*), 652
esp_ble_mesh_cfg_model_app_list_cb_t::company_id (C++ *member*), 652
esp_ble_mesh_cfg_model_app_list_cb_t::element_addr (C++ *member*), 652
esp_ble_mesh_cfg_model_app_list_cb_t::model_id (C++ *member*), 652
esp_ble_mesh_cfg_model_app_list_cb_t::status (C++ *member*), 652
esp_ble_mesh_cfg_model_app_unbind_t (C++ *struct*), 644
esp_ble_mesh_cfg_model_app_unbind_t::company_id (C++ *member*), 645
esp_ble_mesh_cfg_model_app_unbind_t::element_addr (C++ *member*), 644
esp_ble_mesh_cfg_model_app_unbind_t::model_app_id (C++ *member*), 644
esp_ble_mesh_cfg_model_app_unbind_t::model_id (C++ *member*), 645
esp_ble_mesh_cfg_model_pub_get_t (C++ *struct*), 635
esp_ble_mesh_cfg_model_pub_get_t::company_id (C++ *member*), 635
esp_ble_mesh_cfg_model_pub_get_t::element_addr (C++ *member*), 635
esp_ble_mesh_cfg_model_pub_get_t::model_id

(C++ member), 635
esp_ble_mesh_cfg_model_pub_set_t (C++ struct), 639
esp_ble_mesh_cfg_model_pub_set_t::company_id (C++ member), 640
esp_ble_mesh_cfg_model_pub_set_t::cred_flag (C++ member), 639
esp_ble_mesh_cfg_model_pub_set_t::element_addr (C++ member), 639
esp_ble_mesh_cfg_model_pub_set_t::model_id (C++ member), 640
esp_ble_mesh_cfg_model_pub_set_t::publish_addr (C++ member), 639
esp_ble_mesh_cfg_model_pub_set_t::publish_app_idx (C++ member), 639
esp_ble_mesh_cfg_model_pub_set_t::publish_period (C++ member), 640
esp_ble_mesh_cfg_model_pub_set_t::publish_retrans (C++ member), 640
esp_ble_mesh_cfg_model_pub_set_t::publish_ttl (C++ member), 640
esp_ble_mesh_cfg_model_pub_status_cb_t (C++ struct), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::app_idx (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::company_id (C++ member), 648
esp_ble_mesh_cfg_model_pub_status_cb_t::cred_flag (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::element_addr (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::model_id (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::publish_addr (C++ member), 648
esp_ble_mesh_cfg_model_pub_status_cb_t::period (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::publish_retransmit (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::status (C++ member), 647
esp_ble_mesh_cfg_model_pub_status_cb_t::ttl (C++ member), 647
esp_ble_mesh_cfg_model_pub_va_set_t (C++ struct), 642
esp_ble_mesh_cfg_model_pub_va_set_t::company_id (C++ member), 643
esp_ble_mesh_cfg_model_pub_va_set_t::cred_flag (C++ member), 642
esp_ble_mesh_cfg_model_pub_va_set_t::element_addr (C++ member), 642
esp_ble_mesh_cfg_model_pub_va_set_t::label_uuid (C++ member), 642
esp_ble_mesh_cfg_model_pub_va_set_t::model_id (C++ member), 643
esp_ble_mesh_cfg_model_pub_va_set_t::publish_app_idx (C++ member), 642
esp_ble_mesh_cfg_model_pub_va_set_t::publish_period (C++ member), 643
esp_ble_mesh_cfg_model_pub_va_set_t::publish_retransmit (C++ member), 643
esp_ble_mesh_cfg_model_pub_va_set_t::publish_ttl (C++ member), 642
esp_ble_mesh_cfg_model_sub_add_t (C++ struct), 640
esp_ble_mesh_cfg_model_sub_add_t::company_id (C++ member), 640
esp_ble_mesh_cfg_model_sub_add_t::element_addr (C++ member), 640
esp_ble_mesh_cfg_model_sub_add_t::model_id (C++ member), 640
esp_ble_mesh_cfg_model_sub_add_t::sub_addr (C++ member), 640
esp_ble_mesh_cfg_model_sub_delete_all_t (C++ struct), 643
esp_ble_mesh_cfg_model_sub_delete_all_t::company_id (C++ member), 643
esp_ble_mesh_cfg_model_sub_delete_all_t::element_addr (C++ member), 643
esp_ble_mesh_cfg_model_sub_delete_all_t::model_id (C++ member), 643
esp_ble_mesh_cfg_model_sub_delete_t (C++ struct), 640
esp_ble_mesh_cfg_model_sub_delete_t::company_id (C++ member), 641
esp_ble_mesh_cfg_model_sub_delete_t::element_addr (C++ member), 640
esp_ble_mesh_cfg_model_sub_delete_t::model_id (C++ member), 640
esp_ble_mesh_cfg_model_sub_delete_t::sub_addr (C++ member), 640
esp_ble_mesh_cfg_model_sub_list_cb_t (C++ struct), 651
esp_ble_mesh_cfg_model_sub_list_cb_t::company_id (C++ member), 651
esp_ble_mesh_cfg_model_sub_list_cb_t::element_addr (C++ member), 651
esp_ble_mesh_cfg_model_sub_list_cb_t::model_id (C++ member), 651
esp_ble_mesh_cfg_model_sub_list_cb_t::status (C++ member), 651
esp_ble_mesh_cfg_model_sub_list_cb_t::sub_addr (C++ member), 651
esp_ble_mesh_cfg_model_sub_overwrite_t (C++ struct), 641
esp_ble_mesh_cfg_model_sub_overwrite_t::company_id (C++ member), 641
esp_ble_mesh_cfg_model_sub_overwrite_t::element_addr (C++ member), 641
esp_ble_mesh_cfg_model_sub_overwrite_t::model_id (C++ member), 641
esp_ble_mesh_cfg_model_sub_overwrite_t::sub_addr (C++ member), 641
esp_ble_mesh_cfg_model_sub_status_cb_t (C++ struct), 648
esp_ble_mesh_cfg_model_sub_status_cb_t::company_id

(C++ member), 648
esp_ble_mesh_cfg_model_sub_status_cb_t::element_addr (C++ member), 648
esp_ble_mesh_cfg_model_sub_status_cb_t::model_id (C++ member), 648
esp_ble_mesh_cfg_model_sub_status_cb_t::status (C++ member), 648
esp_ble_mesh_cfg_model_sub_status_cb_t::sub_addr (C++ member), 648
esp_ble_mesh_cfg_model_sub_va_add_t (C++ struct), 641
esp_ble_mesh_cfg_model_sub_va_add_t::company_id (C++ member), 641
esp_ble_mesh_cfg_model_sub_va_add_t::element_addr (C++ member), 641
esp_ble_mesh_cfg_model_sub_va_add_t::label_uuid (C++ member), 641
esp_ble_mesh_cfg_model_sub_va_add_t::model_id (C++ member), 641
esp_ble_mesh_cfg_model_sub_va_delete_t (C++ struct), 641
esp_ble_mesh_cfg_model_sub_va_delete_t::company_id (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_delete_t::element_addr (C++ member), 641
esp_ble_mesh_cfg_model_sub_va_delete_t::label_uuid (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_delete_t::model_id (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_overwrite_t (C++ struct), 642
esp_ble_mesh_cfg_model_sub_va_overwrite_t::company_id (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_overwrite_t::element_addr (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_overwrite_t::label_uuid (C++ member), 642
esp_ble_mesh_cfg_model_sub_va_overwrite_t::model_id (C++ member), 642
esp_ble_mesh_cfg_net_key_add_t (C++ struct), 638
esp_ble_mesh_cfg_net_key_add_t::net_idx (C++ member), 638
esp_ble_mesh_cfg_net_key_add_t::net_key (C++ member), 638
esp_ble_mesh_cfg_net_key_delete_t (C++ struct), 643
esp_ble_mesh_cfg_net_key_delete_t::net_idx (C++ member), 643
esp_ble_mesh_cfg_net_key_list_cb_t (C++ struct), 651
esp_ble_mesh_cfg_net_key_list_cb_t::net_idx (C++ member), 651
esp_ble_mesh_cfg_net_key_status_cb_t (C++ struct), 648
esp_ble_mesh_cfg_net_key_status_cb_t::net_idx (C++ member), 648
esp_ble_mesh_cfg_net_key_status_cb_t::status (C++ member), 648
esp_ble_mesh_cfg_net_key_update_t (C++ struct), 643
esp_ble_mesh_cfg_net_key_update_t::net_idx (C++ member), 643
esp_ble_mesh_cfg_net_key_update_t::net_key (C++ member), 643
esp_ble_mesh_cfg_net_trans_status_cb_t (C++ struct), 650
esp_ble_mesh_cfg_net_trans_status_cb_t::net_trans (C++ member), 651
esp_ble_mesh_cfg_net_trans_status_cb_t::net_trans (C++ member), 651
esp_ble_mesh_cfg_net_transmit_set_t (C++ struct), 645
esp_ble_mesh_cfg_net_transmit_set_t::net_transmit (C++ member), 645
esp_ble_mesh_cfg_node_id_status_cb_t (C++ struct), 652
esp_ble_mesh_cfg_node_id_status_cb_t::identity (C++ member), 652
esp_ble_mesh_cfg_node_id_status_cb_t::net_idx (C++ member), 652
esp_ble_mesh_cfg_node_id_status_cb_t::status (C++ member), 652
esp_ble_mesh_cfg_node_identity_get_t (C++ struct), 636
esp_ble_mesh_cfg_node_identity_get_t::net_idx (C++ member), 636
esp_ble_mesh_cfg_node_identity_set_t (C++ struct), 644
esp_ble_mesh_cfg_node_identity_set_t::identity (C++ member), 644
esp_ble_mesh_cfg_node_identity_set_t::net_idx (C++ member), 644
esp_ble_mesh_cfg_relay_set_t (C++ struct), 638
esp_ble_mesh_cfg_relay_set_t::relay (C++ member), 638
esp_ble_mesh_cfg_relay_set_t::relay_retransmit (C++ member), 638
esp_ble_mesh_cfg_relay_status_cb_t (C++ struct), 647
esp_ble_mesh_cfg_relay_status_cb_t::relay (C++ member), 647
esp_ble_mesh_cfg_relay_status_cb_t::retransmit (C++ member), 647
esp_ble_mesh_cfg_server_cb_event_t (C++ enum), 658
esp_ble_mesh_cfg_server_cb_event_t::ESP_BLE_MESH_ (C++ enumerator), 659
esp_ble_mesh_cfg_server_cb_event_t::ESP_BLE_MESH_ (C++ enumerator), 658
esp_ble_mesh_cfg_server_cb_param_t (C++ struct), 657
esp_ble_mesh_cfg_server_cb_param_t::ctx (C++ member), 657
esp_ble_mesh_cfg_server_cb_param_t::model

(C++ member), 657
esp_ble_mesh_cfg_server_cb_param_t::value (C++ member), 657
esp_ble_mesh_cfg_server_cb_t (C++ type), 658
esp_ble_mesh_cfg_server_cb_value_t (C++ union), 633
esp_ble_mesh_cfg_server_cb_value_t::state_change (C++ member), 634
esp_ble_mesh_cfg_server_state_change_t (C++ union), 633
esp_ble_mesh_cfg_server_state_change_t::appkey_add (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::appkey_delete (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::appkey_update (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::kr_phase_set (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::mod_app_bind (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::mod_app_unbind (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::mod_pub_set (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::mod_sub_add (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::mod_sub_delete (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::netkey_add (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::netkey_delete (C++ member), 633
esp_ble_mesh_cfg_server_state_change_t::netkey_update (C++ member), 633
esp_ble_mesh_cfg_sig_model_app_get_t (C++ struct), 636
esp_ble_mesh_cfg_sig_model_app_get_t::element_addr (C++ member), 637
esp_ble_mesh_cfg_sig_model_app_get_t::model_id (C++ member), 637
esp_ble_mesh_cfg_sig_model_sub_get_t (C++ struct), 636
esp_ble_mesh_cfg_sig_model_sub_get_t::element_addr (C++ member), 636
esp_ble_mesh_cfg_sig_model_sub_get_t::model_id (C++ member), 636
esp_ble_mesh_cfg_srv (C++ struct), 634
esp_ble_mesh_cfg_srv::beacon (C++ member), 634
esp_ble_mesh_cfg_srv::count (C++ member), 634
esp_ble_mesh_cfg_srv::default_ttl (C++ member), 634
esp_ble_mesh_cfg_srv::dst (C++ member), 634
esp_ble_mesh_cfg_srv::expiry (C++ member), 635
esp_ble_mesh_cfg_srv::feature (C++ member), 635
esp_ble_mesh_cfg_srv::friend_state (C++ member), 634
esp_ble_mesh_cfg_srv::gatt_proxy (C++ member), 634
esp_ble_mesh_cfg_srv::heartbeat_pub (C++ member), 635
esp_ble_mesh_cfg_srv::heartbeat_recv_cb (C++ member), 635
esp_ble_mesh_cfg_srv::heartbeat_sub (C++ member), 635
esp_ble_mesh_cfg_srv::max_hops (C++ member), 635
esp_ble_mesh_cfg_srv::min_hops (C++ member), 635
esp_ble_mesh_cfg_srv::model (C++ member), 634
esp_ble_mesh_cfg_srv::net_idx (C++ member), 635
esp_ble_mesh_cfg_srv::net_transmit (C++ member), 634
esp_ble_mesh_cfg_srv::period (C++ member), 634
esp_ble_mesh_cfg_srv::relay (C++ member), 634
esp_ble_mesh_cfg_srv::relay_retransmit (C++ member), 634
esp_ble_mesh_cfg_srv::src (C++ member), 635
esp_ble_mesh_cfg_srv::timer (C++ member), 634
esp_ble_mesh_cfg_srv::ttl (C++ member), 634
esp_ble_mesh_cfg_srv_t (C++ type), 658
ESP_BLE_MESH_CFG_STATUS_CANNOT_BIND (C macro), 586
ESP_BLE_MESH_CFG_STATUS_CANNOT_REMOVE (C macro), 586
ESP_BLE_MESH_CFG_STATUS_CANNOT_SET (C macro), 586
ESP_BLE_MESH_CFG_STATUS_CANNOT_UPDATE (C macro), 586
ESP_BLE_MESH_CFG_STATUS_FEATURE_NOT_SUPPORTED (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INSUFFICIENT_RESOURCES (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INVALID_ADDRESS (C macro), 585
ESP_BLE_MESH_CFG_STATUS_INVALID_APPKEY (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INVALID_BINDING (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INVALID_MODEL (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INVALID_NETKEY (C macro), 586
ESP_BLE_MESH_CFG_STATUS_INVALID_PUBLISH_PARAMETER

(*C macro*), 586
ESP_BLE_MESH_CFG_STATUS_KEY_INDEX_ALREADY_STORED (*C macro*), 586
ESP_BLE_MESH_CFG_STATUS_NOT_A_SUBSCRIBE_MODEL (*C macro*), 586
ESP_BLE_MESH_CFG_STATUS_STORAGE_FAILURE (*C macro*), 586
ESP_BLE_MESH_CFG_STATUS_SUCCESS (*C macro*), 585
esp_ble_mesh_cfg_status_t (*C++ type*), 597
ESP_BLE_MESH_CFG_STATUS_TEMP_UNABLE_TO_CHANGE_STATE (*C macro*), 586
ESP_BLE_MESH_CFG_STATUS_UNSPECIFIED_ERROR (*C macro*), 586
esp_ble_mesh_cfg_vnd_model_app_get_t (*C++ struct*), 637
esp_ble_mesh_cfg_vnd_model_app_get_t::company_id (*C++ member*), 637
esp_ble_mesh_cfg_vnd_model_app_get_t::element_addr (*C++ member*), 637
esp_ble_mesh_cfg_vnd_model_app_get_t::model_id (*C++ member*), 637
esp_ble_mesh_cfg_vnd_model_sub_get_t (*C++ struct*), 636
esp_ble_mesh_cfg_vnd_model_sub_get_t::company_id (*C++ member*), 636
esp_ble_mesh_cfg_vnd_model_sub_get_t::element_addr (*C++ member*), 636
esp_ble_mesh_cfg_vnd_model_sub_get_t::model_id (*C++ member*), 636
ESP_BLE_MESH_CID_NVAL (*C macro*), 574
esp_ble_mesh_client_common_param_t (*C++ struct*), 572
esp_ble_mesh_client_common_param_t::ctx (*C++ member*), 572
esp_ble_mesh_client_common_param_t::model (*C++ member*), 572
esp_ble_mesh_client_common_param_t::msg_role (*C++ member*), 572
esp_ble_mesh_client_common_param_t::msg_timeout (*C++ member*), 572
esp_ble_mesh_client_common_param_t::opcode (*C++ member*), 572
esp_ble_mesh_client_model_deinit (*C++ function*), 613
esp_ble_mesh_client_model_init (*C++ function*), 612
esp_ble_mesh_client_model_send_msg (*C++ function*), 613
esp_ble_mesh_client_op_pair_t (*C++ struct*), 571
esp_ble_mesh_client_op_pair_t::cli_op (*C++ member*), 571
esp_ble_mesh_client_op_pair_t::status_op (*C++ member*), 571
esp_ble_mesh_client_t (*C++ struct*), 571
esp_ble_mesh_client_t::internal_data (*C++ member*), 571
esp_ble_mesh_client_t::model (*C++ member*), 571
esp_ble_mesh_client_t::msg_role (*C++ member*), 571
esp_ble_mesh_client_t::op_pair (*C++ member*), 571
esp_ble_mesh_client_t::op_pair_size (*C++ member*), 571
esp_ble_mesh_client_t::publish_status (*C++ member*), 571
esp_ble_mesh_comp_t (*C++ struct*), 567
esp_ble_mesh_comp_t::cid (*C++ member*), 568
esp_ble_mesh_comp_t::element_count (*C++ member*), 568
esp_ble_mesh_comp_t::elements (*C++ member*), 568
esp_ble_mesh_comp_t::pid (*C++ member*), 568
esp_ble_mesh_comp_t::vid (*C++ member*), 568
ESP_BLE_MESH_CONDENSATION_ERROR (*C macro*), 667
ESP_BLE_MESH_CONDENSATION_WARNING (*C macro*), 667
esp_ble_mesh_config_client_get_state (*C++ function*), 628
esp_ble_mesh_config_client_set_state (*C++ function*), 628
ESP_BLE_MESH_CONFIGURATION_ERROR (*C macro*), 667
ESP_BLE_MESH_CONFIGURATION_WARNING (*C macro*), 667
esp_ble_mesh_deinit (*C++ function*), 608
esp_ble_mesh_deinit_param_t (*C++ struct*), 563
esp_ble_mesh_deinit_param_t::erase_flash (*C++ member*), 563
esp_ble_mesh_dev_add_flag_t (*C++ type*), 597
esp_ble_mesh_dev_role_t (*C++ enum*), 600
esp_ble_mesh_dev_role_t::ROLE_FAST_PROV (*C++ enumerator*), 600
esp_ble_mesh_dev_role_t::ROLE_NODE (*C++ enumerator*), 600
esp_ble_mesh_dev_role_t::ROLE_PROVISIONER (*C++ enumerator*), 600
esp_ble_mesh_device_delete_t (*C++ struct*), 568
esp_ble_mesh_device_delete_t::addr (*C++ member*), 568
esp_ble_mesh_device_delete_t::addr_type (*C++ member*), 568
esp_ble_mesh_device_delete_t::flag (*C++ member*), 569
esp_ble_mesh_device_delete_t::uuid (*C++ member*), 568
ESP_BLE_MESH_DEVICE_DROPPED_ERROR (*C*

macro), 668
ESP_BLE_MESH_DEVICE_DROPPED_WARNING (*C macro*), 668
ESP_BLE_MESH_DEVICE_MOVED_ERROR (*C macro*), 668
ESP_BLE_MESH_DEVICE_MOVED_WARNING (*C macro*), 668
ESP_BLE_MESH_DEVICE_NAME_MAX_LEN (*C macro*), 574
esp_ble_mesh_elem_t (*C++ struct*), 563
esp_ble_mesh_elem_t::element_addr (*C++ member*), 563
esp_ble_mesh_elem_t::location (*C++ member*), 563
esp_ble_mesh_elem_t::sig_model_count (*C++ member*), 564
esp_ble_mesh_elem_t::sig_models (*C++ member*), 564
esp_ble_mesh_elem_t::vnd_model_count (*C++ member*), 564
esp_ble_mesh_elem_t::vnd_models (*C++ member*), 564
ESP_BLE_MESH_ELEMENT (*C macro*), 577
ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_ERROR (*C macro*), 668
ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_WARNING (*C macro*), 667
ESP_BLE_MESH_EMPTY_ERROR (*C macro*), 669
ESP_BLE_MESH_EMPTY_WARNING (*C macro*), 669
esp_ble_mesh_fast_prov_action_t (*C++ enum*), 600
esp_ble_mesh_fast_prov_action_t::FAST_PROV_ACT_ENTER_MESH (*C++ enumerator*), 601
esp_ble_mesh_fast_prov_action_t::FAST_PROV_ACT_EXIT_MESH (*C++ enumerator*), 601
esp_ble_mesh_fast_prov_action_t::FAST_PROV_ACT_MAX (*C++ enumerator*), 601
esp_ble_mesh_fast_prov_action_t::FAST_PROV_ACT_NONE (*C++ enumerator*), 600
esp_ble_mesh_fast_prov_action_t::FAST_PROV_ACT_SUSPEND_MESH (*C++ enumerator*), 601
esp_ble_mesh_fast_prov_info_t (*C++ struct*), 570
esp_ble_mesh_fast_prov_info_t::flags (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::iv_index (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::match_len (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::match_val (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::net_idx (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::offset (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::unicast_max (*C++ member*), 570
esp_ble_mesh_fast_prov_info_t::unicast_min (*C++ member*), 570
ESP_BLE_MESH_FEATURE_ALL_SUPPORTED (*C macro*), 576
ESP_BLE_MESH_FEATURE_FRIEND (*C macro*), 576
ESP_BLE_MESH_FEATURE_LOW_POWER (*C macro*), 576
ESP_BLE_MESH_FEATURE_PROXY (*C macro*), 576
ESP_BLE_MESH_FEATURE_RELAY (*C macro*), 576
esp_ble_mesh_find_element (*C++ function*), 609
esp_ble_mesh_find_sig_model (*C++ function*), 609
esp_ble_mesh_find_vendor_model (*C++ function*), 609
ESP_BLE_MESH_FRIEND_DISABLED (*C macro*), 576
ESP_BLE_MESH_FRIEND_ENABLED (*C macro*), 576
ESP_BLE_MESH_FRIEND_NOT_SUPPORTED (*C macro*), 576
ESP_BLE_MESH_GATT_PROXY_DISABLED (*C macro*), 575
ESP_BLE_MESH_GATT_PROXY_ENABLED (*C macro*), 575
ESP_BLE_MESH_GATT_PROXY_NOT_SUPPORTED (*C macro*), 575
esp_ble_mesh_gen_admin_prop_access_t (*C++ enum*), 709
esp_ble_mesh_gen_admin_prop_access_t::ESP_BLE_MESH_GEN_ADMIN_PROP_ACCESS_ADMIN (*C++ enumerator*), 709
esp_ble_mesh_gen_admin_prop_access_t::ESP_BLE_MESH_GEN_ADMIN_PROP_ACCESS_MANUFACTURER (*C++ enumerator*), 710
esp_ble_mesh_gen_admin_prop_access_t::ESP_BLE_MESH_GEN_ADMIN_PROP_ACCESS_NONE (*C++ enumerator*), 710
esp_ble_mesh_gen_admin_prop_access_t::ESP_BLE_MESH_GEN_ADMIN_PROP_ACCESS_USER (*C++ enumerator*), 709
esp_ble_mesh_gen_admin_prop_srv_t (*C++ struct*), 693
esp_ble_mesh_gen_admin_prop_srv_t::model (*C++ member*), 694
esp_ble_mesh_gen_admin_prop_srv_t::properties (*C++ member*), 694
esp_ble_mesh_gen_admin_prop_srv_t::property_count (*C++ member*), 694
esp_ble_mesh_gen_admin_prop_srv_t::rsp_ctrl (*C++ member*), 694
esp_ble_mesh_gen_admin_properties_status_cb_t (*C++ struct*), 685
esp_ble_mesh_gen_admin_properties_status_cb_t::properties (*C++ member*), 685
esp_ble_mesh_gen_admin_property_get_t (*C++ struct*), 680
esp_ble_mesh_gen_admin_property_get_t::property_id (*C++ member*), 680
esp_ble_mesh_gen_admin_property_set_t (*C++ struct*), 680
esp_ble_mesh_gen_admin_property_set_t::property_id

(C++ *member*), 681
esp_ble_mesh_gen_admin_property_set_t::property_value (C++ *member*), 681
esp_ble_mesh_gen_admin_property_set_t::user_access (C++ *member*), 681
esp_ble_mesh_gen_admin_property_status_cb_t (C++ *struct*), 685
esp_ble_mesh_gen_admin_property_status_cb_t::admin_property_id (C++ *member*), 685
esp_ble_mesh_gen_admin_property_status_cb_t::property_id (C++ *member*), 685
esp_ble_mesh_gen_admin_property_status_cb_t::property_value (C++ *member*), 685
esp_ble_mesh_gen_admin_property_status_cb_t::user_access (C++ *member*), 685
esp_ble_mesh_gen_battery_srv_t (C++ *struct*), 691
esp_ble_mesh_gen_battery_srv_t::model (C++ *member*), 691
esp_ble_mesh_gen_battery_srv_t::rsp_ctrl (C++ *member*), 691
esp_ble_mesh_gen_battery_srv_t::state (C++ *member*), 691
esp_ble_mesh_gen_battery_state_t (C++ *struct*), 691
esp_ble_mesh_gen_battery_state_t::battery_flags (C++ *member*), 691
esp_ble_mesh_gen_battery_state_t::battery_level (C++ *member*), 691
esp_ble_mesh_gen_battery_state_t::time_to_charge (C++ *member*), 691
esp_ble_mesh_gen_battery_state_t::time_to_discharge (C++ *member*), 691
esp_ble_mesh_gen_battery_status_cb_t (C++ *struct*), 683
esp_ble_mesh_gen_battery_status_cb_t::battery_level (C++ *member*), 684
esp_ble_mesh_gen_battery_status_cb_t::flags (C++ *member*), 684
esp_ble_mesh_gen_battery_status_cb_t::time_to_charge (C++ *member*), 684
esp_ble_mesh_gen_battery_status_cb_t::time_to_discharge (C++ *member*), 684
esp_ble_mesh_gen_client_prop_srv_t (C++ *struct*), 694
esp_ble_mesh_gen_client_prop_srv_t::id (C++ *member*), 694
esp_ble_mesh_gen_client_prop_srv_t::model (C++ *member*), 694
esp_ble_mesh_gen_client_prop_srv_t::property_id (C++ *member*), 695
esp_ble_mesh_gen_client_prop_srv_t::rsp_ctrl (C++ *member*), 694
esp_ble_mesh_gen_client_properties_get_t (C++ *struct*), 681
esp_ble_mesh_gen_client_properties_get_t::property_id (C++ *member*), 681
esp_ble_mesh_gen_client_properties_status_cb_t (C++ *struct*), 686
esp_ble_mesh_gen_client_properties_status_cb_t::property_ids (C++ *member*), 686
esp_ble_mesh_gen_client_status_cb_t (C++ *union*), 672
esp_ble_mesh_gen_client_status_cb_t::admin_property_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::admin_property_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::battery_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::client_properties_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::def_trans_time_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::level_status (C++ *member*), 672
esp_ble_mesh_gen_client_status_cb_t::location_global_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::location_local_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::manufacturer_properties (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::manufacturer_property (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::onoff_status (C++ *member*), 672
esp_ble_mesh_gen_client_status_cb_t::onpowerup_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::power_default_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::power_last_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::power_level_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::power_range_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::user_properties_status (C++ *member*), 673
esp_ble_mesh_gen_client_status_cb_t::user_property_status (C++ *member*), 673
esp_ble_mesh_gen_def_trans_time_set_t (C++ *struct*), 678
esp_ble_mesh_gen_def_trans_time_set_t::trans_time (C++ *member*), 678
esp_ble_mesh_gen_def_trans_time_srv_t (C++ *struct*), 688
esp_ble_mesh_gen_def_trans_time_srv_t::model (C++ *member*), 689
esp_ble_mesh_gen_def_trans_time_srv_t::rsp_ctrl (C++ *member*), 689
esp_ble_mesh_gen_def_trans_time_srv_t::state (C++ *member*), 689
esp_ble_mesh_gen_def_trans_time_state_t (C++ *struct*), 688
esp_ble_mesh_gen_def_trans_time_state_t::trans_time (C++ *member*), 688
esp_ble_mesh_gen_def_trans_time_status_cb_t

(C++ *struct*), 682
esp_ble_mesh_gen_def_trans_time_status_cb_t::trans_time (C++ *member*), 682
esp_ble_mesh_gen_delta_set_t (C++ *struct*), 677
esp_ble_mesh_gen_delta_set_t::delay (C++ *member*), 677
esp_ble_mesh_gen_delta_set_t::level (C++ *member*), 677
esp_ble_mesh_gen_delta_set_t::op_en (C++ *member*), 677
esp_ble_mesh_gen_delta_set_t::tid (C++ *member*), 677
esp_ble_mesh_gen_delta_set_t::trans_time (C++ *member*), 677
esp_ble_mesh_gen_level_set_t (C++ *struct*), 677
esp_ble_mesh_gen_level_set_t::delay (C++ *member*), 677
esp_ble_mesh_gen_level_set_t::level (C++ *member*), 677
esp_ble_mesh_gen_level_set_t::op_en (C++ *member*), 677
esp_ble_mesh_gen_level_set_t::tid (C++ *member*), 677
esp_ble_mesh_gen_level_set_t::trans_time (C++ *member*), 677
esp_ble_mesh_gen_level_srv_t (C++ *struct*), 688
esp_ble_mesh_gen_level_srv_t::last (C++ *member*), 688
esp_ble_mesh_gen_level_srv_t::model (C++ *member*), 688
esp_ble_mesh_gen_level_srv_t::rsp_ctrl (C++ *member*), 688
esp_ble_mesh_gen_level_srv_t::state (C++ *member*), 688
esp_ble_mesh_gen_level_srv_t::transition (C++ *member*), 688
esp_ble_mesh_gen_level_srv_t::tt_delta_level (C++ *member*), 688
esp_ble_mesh_gen_level_state_t (C++ *struct*), 687
esp_ble_mesh_gen_level_state_t::last_delta (C++ *member*), 688
esp_ble_mesh_gen_level_state_t::last_level (C++ *member*), 687
esp_ble_mesh_gen_level_state_t::level (C++ *member*), 687
esp_ble_mesh_gen_level_state_t::move_start (C++ *member*), 688
esp_ble_mesh_gen_level_state_t::positive (C++ *member*), 688
esp_ble_mesh_gen_level_state_t::target_level (C++ *member*), 687
esp_ble_mesh_gen_level_status_cb_t (C++ *struct*), 682
esp_ble_mesh_gen_level_status_cb_t::op_en (C++ *member*), 682
esp_ble_mesh_gen_level_status_cb_t::present_level (C++ *member*), 682
esp_ble_mesh_gen_level_status_cb_t::remain_time (C++ *member*), 682
esp_ble_mesh_gen_level_status_cb_t::target_level (C++ *member*), 682
esp_ble_mesh_gen_loc_global_set_t (C++ *struct*), 679
esp_ble_mesh_gen_loc_global_set_t::global_altitud (C++ *member*), 679
esp_ble_mesh_gen_loc_global_set_t::global_latitud (C++ *member*), 679
esp_ble_mesh_gen_loc_global_set_t::global_longitu (C++ *member*), 679
esp_ble_mesh_gen_loc_global_status_cb_t (C++ *struct*), 684
esp_ble_mesh_gen_loc_global_status_cb_t::global_a (C++ *member*), 684
esp_ble_mesh_gen_loc_global_status_cb_t::global_l (C++ *member*), 684
esp_ble_mesh_gen_loc_global_status_cb_t::global_l (C++ *member*), 684
esp_ble_mesh_gen_loc_local_set_t (C++ *struct*), 679
esp_ble_mesh_gen_loc_local_set_t::floor_number (C++ *member*), 680
esp_ble_mesh_gen_loc_local_set_t::local_altitude (C++ *member*), 680
esp_ble_mesh_gen_loc_local_set_t::local_east (C++ *member*), 680
esp_ble_mesh_gen_loc_local_set_t::local_north (C++ *member*), 680
esp_ble_mesh_gen_loc_local_set_t::uncertainty (C++ *member*), 680
esp_ble_mesh_gen_loc_local_status_cb_t (C++ *struct*), 684
esp_ble_mesh_gen_loc_local_status_cb_t::floor_num (C++ *member*), 684
esp_ble_mesh_gen_loc_local_status_cb_t::local_alt (C++ *member*), 684
esp_ble_mesh_gen_loc_local_status_cb_t::local_eas (C++ *member*), 684
esp_ble_mesh_gen_loc_local_status_cb_t::local_nor (C++ *member*), 684
esp_ble_mesh_gen_loc_local_status_cb_t::uncertain (C++ *member*), 684
esp_ble_mesh_gen_location_setup_srv_t (C++ *struct*), 692
esp_ble_mesh_gen_location_setup_srv_t::model (C++ *member*), 693
esp_ble_mesh_gen_location_setup_srv_t::rsp_ctrl (C++ *member*), 693
esp_ble_mesh_gen_location_setup_srv_t::state (C++ *member*), 693
esp_ble_mesh_gen_location_srv_t (C++ *struct*), 692
esp_ble_mesh_gen_location_srv_t::model

(C++ *member*), 692
esp_ble_mesh_gen_location_srv_t::rsp_ctrl (C++ *member*), 692
esp_ble_mesh_gen_location_srv_t::state (C++ *member*), 692
esp_ble_mesh_gen_location_state_t (C++ *struct*), 692
esp_ble_mesh_gen_location_state_t::floor_number (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::global_altitude (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::global_latitude (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::global_longitude (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::local_altitude (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::local_east (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::local_north (C++ *member*), 692
esp_ble_mesh_gen_location_state_t::uncertainty (C++ *member*), 692
esp_ble_mesh_gen_manu_prop_access_t (C++ *enum*), 710
esp_ble_mesh_gen_manu_prop_access_t::ESP_BLE_MESH_GEN_MANU_ACCESS_READ (C++ *enumerator*), 710
esp_ble_mesh_gen_manu_prop_access_t::ESP_BLE_MESH_GEN_MANU_ACCESS_NOT_USER_PROP (C++ *enumerator*), 710
esp_ble_mesh_gen_manu_prop_srv_t (C++ *struct*), 694
esp_ble_mesh_gen_manu_prop_srv_t::model (C++ *member*), 694
esp_ble_mesh_gen_manu_prop_srv_t::properties (C++ *member*), 694
esp_ble_mesh_gen_manu_prop_srv_t::property_id (C++ *member*), 694
esp_ble_mesh_gen_manu_prop_srv_t::rsp_ctrl (C++ *member*), 694
esp_ble_mesh_gen_manufacturer_properties_state_t (C++ *struct*), 686
esp_ble_mesh_gen_manufacturer_properties_status_cb_t (C++ *member*), 686
esp_ble_mesh_gen_manufacturer_property_get_t (C++ *struct*), 681
esp_ble_mesh_gen_manufacturer_property_get_t::property_id (C++ *member*), 681
esp_ble_mesh_gen_manufacturer_property_set_t (C++ *struct*), 681
esp_ble_mesh_gen_manufacturer_property_set_t::property_id (C++ *member*), 681
esp_ble_mesh_gen_manufacturer_property_set_t::user_access (C++ *member*), 681
esp_ble_mesh_gen_manufacturer_property_state_t (C++ *struct*), 686
esp_ble_mesh_gen_manufacturer_property_state_t::property_id (C++ *member*), 686
esp_ble_mesh_gen_manufacturer_property_state_t::property_len (C++ *member*), 686
esp_ble_mesh_gen_manufacturer_property_state_t::user_access (C++ *member*), 686
esp_ble_mesh_gen_manufacturer_property_status_cb_t (C++ *struct*), 686
esp_ble_mesh_gen_manufacturer_property_status_cb_t::op_en (C++ *member*), 686
esp_ble_mesh_gen_manufacturer_property_status_cb_t::property_id (C++ *member*), 686
esp_ble_mesh_gen_move_set_t (C++ *struct*), 678
esp_ble_mesh_gen_move_set_t::delay (C++ *member*), 678
esp_ble_mesh_gen_move_set_t::delta_level (C++ *member*), 678
esp_ble_mesh_gen_move_set_t::op_en (C++ *member*), 678
esp_ble_mesh_gen_move_set_t::tid (C++ *member*), 678
esp_ble_mesh_gen_move_set_t::trans_time (C++ *member*), 678
esp_ble_mesh_gen_onoff_set_t (C++ *struct*), 676
esp_ble_mesh_gen_onoff_set_t::delay (C++ *member*), 677
esp_ble_mesh_gen_onoff_set_t::onoff (C++ *member*), 676
esp_ble_mesh_gen_onoff_set_t::op_en (C++ *member*), 676
esp_ble_mesh_gen_onoff_set_t::tid (C++ *member*), 677
esp_ble_mesh_gen_onoff_set_t::trans_time (C++ *member*), 677
esp_ble_mesh_gen_onoff_srv_t (C++ *struct*), 687
esp_ble_mesh_gen_onoff_srv_t::last (C++ *member*), 687
esp_ble_mesh_gen_onoff_srv_t::model (C++ *member*), 687
esp_ble_mesh_gen_onoff_srv_t::rsp_ctrl (C++ *member*), 687
esp_ble_mesh_gen_onoff_srv_t::state (C++ *member*), 687
esp_ble_mesh_gen_onoff_srv_t::transition (C++ *member*), 687
esp_ble_mesh_gen_onoff_srv_t::onoff_state_t (C++ *struct*), 687
esp_ble_mesh_gen_onoff_state_t::onoff (C++ *member*), 687
esp_ble_mesh_gen_onoff_state_t::target_onoff (C++ *member*), 687
esp_ble_mesh_gen_onoff_status_cb_t (C++ *struct*), 681
esp_ble_mesh_gen_onoff_status_cb_t::op_en (C++ *member*), 681
esp_ble_mesh_gen_onoff_status_cb_t::present_onoff (C++ *member*), 682
esp_ble_mesh_gen_onoff_status_cb_t::remain_time (C++ *member*), 682
esp_ble_mesh_gen_onoff_status_cb_t::target_onoff (C++ *member*), 682
esp_ble_mesh_gen_property_low_up_set_t (C++

struct), 678
esp_ble_mesh_gen_onpowerup_set_t::onpowerup (*C++ member*), 678
esp_ble_mesh_gen_onpowerup_state_t (*C++ struct*), 689
esp_ble_mesh_gen_onpowerup_state_t::onpowerup (*C++ member*), 689
esp_ble_mesh_gen_onpowerup_status_cb_t (*C++ struct*), 682
esp_ble_mesh_gen_onpowerup_status_cb_t::onpowerup (*C++ member*), 682
esp_ble_mesh_gen_power_default_set_t (*C++ struct*), 679
esp_ble_mesh_gen_power_default_set_t::power (*C++ member*), 679
esp_ble_mesh_gen_power_default_status_cb_t (*C++ struct*), 683
esp_ble_mesh_gen_power_default_status_cb_t::power (*C++ member*), 683
esp_ble_mesh_gen_power_last_status_cb_t (*C++ struct*), 683
esp_ble_mesh_gen_power_last_status_cb_t::power (*C++ member*), 683
esp_ble_mesh_gen_power_level_set_t (*C++ struct*), 678
esp_ble_mesh_gen_power_level_set_t::delay (*C++ member*), 679
esp_ble_mesh_gen_power_level_set_t::op_en (*C++ member*), 678
esp_ble_mesh_gen_power_level_set_t::power (*C++ member*), 678
esp_ble_mesh_gen_power_level_set_t::tid (*C++ member*), 679
esp_ble_mesh_gen_power_level_set_t::trans_time (*C++ member*), 679
esp_ble_mesh_gen_power_level_setup_srv_t (*C++ struct*), 691
esp_ble_mesh_gen_power_level_setup_srv_t::model (*C++ member*), 691
esp_ble_mesh_gen_power_level_setup_srv_t::rsp_ctrl (*C++ member*), 691
esp_ble_mesh_gen_power_level_setup_srv_t::state (*C++ member*), 691
esp_ble_mesh_gen_power_level_srv_t (*C++ struct*), 690
esp_ble_mesh_gen_power_level_srv_t::last (*C++ member*), 690
esp_ble_mesh_gen_power_level_srv_t::model (*C++ member*), 690
esp_ble_mesh_gen_power_level_srv_t::rsp_ctrl (*C++ member*), 690
esp_ble_mesh_gen_power_level_srv_t::state (*C++ member*), 690
esp_ble_mesh_gen_power_level_srv_t::trans_time (*C++ member*), 690
esp_ble_mesh_gen_power_level_srv_t::tt_delta_level (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t (*C++ struct*), 689
esp_ble_mesh_gen_power_level_state_t::power_actual (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::power_default (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::power_last (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::power_range_max (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::power_range_min (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::status_code (*C++ member*), 690
esp_ble_mesh_gen_power_level_state_t::target_power (*C++ member*), 690
esp_ble_mesh_gen_power_level_status_cb_t (*C++ struct*), 682
esp_ble_mesh_gen_power_level_status_cb_t::op_en (*C++ member*), 683
esp_ble_mesh_gen_power_level_status_cb_t::present (*C++ member*), 683
esp_ble_mesh_gen_power_level_status_cb_t::remain (*C++ member*), 683
esp_ble_mesh_gen_power_level_status_cb_t::target (*C++ member*), 683
esp_ble_mesh_gen_power_onoff_setup_srv_t (*C++ struct*), 689
esp_ble_mesh_gen_power_onoff_setup_srv_t::model (*C++ member*), 689
esp_ble_mesh_gen_power_onoff_setup_srv_t::rsp_ctrl (*C++ member*), 689
esp_ble_mesh_gen_power_onoff_setup_srv_t::state (*C++ member*), 689
esp_ble_mesh_gen_power_onoff_srv_t (*C++ struct*), 689
esp_ble_mesh_gen_power_onoff_srv_t::model (*C++ member*), 689
esp_ble_mesh_gen_power_onoff_srv_t::rsp_ctrl (*C++ member*), 689
esp_ble_mesh_gen_power_onoff_srv_t::state (*C++ member*), 689
esp_ble_mesh_gen_power_range_set_t (*C++ struct*), 679
esp_ble_mesh_gen_power_range_set_t::range_max (*C++ member*), 679
esp_ble_mesh_gen_power_range_set_t::range_min (*C++ member*), 679
esp_ble_mesh_gen_power_range_status_cb_t (*C++ struct*), 683
esp_ble_mesh_gen_power_range_status_cb_t::range_max (*C++ member*), 683
esp_ble_mesh_gen_power_range_status_cb_t::range_min (*C++ member*), 683
esp_ble_mesh_gen_power_range_status_cb_t::status_code (*C++ member*), 683
esp_ble_mesh_gen_user_prop_access_t (*C++ enum*), 709
esp_ble_mesh_gen_user_prop_access_t::ESP_BLE_MESH

(C++ *enumerator*), 709
esp_ble_mesh_gen_user_prop_access_t::ESP_BLE_MESH_GEN_USER_ACCESS_READ (C++ *enumerator*), 709
esp_ble_mesh_gen_user_prop_access_t::ESP_BLE_MESH_GEN_USER_ACCESS_READ_WRITE (C++ *enumerator*), 709
esp_ble_mesh_gen_user_prop_access_t::ESP_BLE_MESH_GEN_USER_ACCESS_WRITE (C++ *enumerator*), 709
esp_ble_mesh_gen_user_prop_srv_t (C++ *struct*), 693
esp_ble_mesh_gen_user_prop_srv_t::model (C++ *member*), 693
esp_ble_mesh_gen_user_prop_srv_t::properties (C++ *member*), 693
esp_ble_mesh_gen_user_prop_srv_t::property_count (C++ *member*), 693
esp_ble_mesh_gen_user_prop_srv_t::rsp_ctrl (C++ *member*), 693
esp_ble_mesh_gen_user_properties_status_cb (C++ *struct*), 684
esp_ble_mesh_gen_user_properties_status_cb::properties (C++ *member*), 685
esp_ble_mesh_gen_user_property_get_t (C++ *struct*), 680
esp_ble_mesh_gen_user_property_get_t::property_id (C++ *member*), 680
esp_ble_mesh_gen_user_property_set_t (C++ *struct*), 680
esp_ble_mesh_gen_user_property_set_t::property_id (C++ *member*), 680
esp_ble_mesh_gen_user_property_set_t::property_value (C++ *member*), 680
esp_ble_mesh_gen_user_property_status_cb (C++ *struct*), 685
esp_ble_mesh_gen_user_property_status_cb::admin_type (C++ *member*), 685
esp_ble_mesh_gen_user_property_status_cb::property_id (C++ *member*), 685
esp_ble_mesh_gen_user_property_status_cb::property_len (C++ *member*), 685
esp_ble_mesh_gen_user_property_status_cb::user_access (C++ *member*), 685
esp_ble_mesh_generic_client_cb_event_t (C++ *enum*), 709
esp_ble_mesh_generic_client_cb_event_t::ESP_BLE_MESH_GENERIC_CLIENT_EVT_MAX (C++ *enumerator*), 709
esp_ble_mesh_generic_client_cb_event_t::ESP_BLE_MESH_GENERIC_CLIENT_GET_STATE_EVT (C++ *enumerator*), 709
esp_ble_mesh_generic_client_cb_event_t::ESP_BLE_MESH_GENERIC_CLIENT_PUBLISH_EVT (C++ *enumerator*), 709
esp_ble_mesh_generic_client_cb_event_t::ESP_BLE_MESH_GENERIC_CLIENT_SET_STATE_EVT (C++ *enumerator*), 709
esp_ble_mesh_generic_client_cb_event_t::ESP_BLE_MESH_GENERIC_CLIENT_TIMEOUT_EVT (C++ *enumerator*), 709
esp_ble_mesh_generic_client_cb_param_t (C++ *struct*), 686
esp_ble_mesh_generic_client_cb_param_t::error_code (C++ *member*), 686
esp_ble_mesh_generic_client_cb_param_t::params (C++ *member*), 686
esp_ble_mesh_generic_client_cb_param_t::status_cb (C++ *member*), 686
esp_ble_mesh_generic_client_cb_param_t::type (C++ *type*), 708
esp_ble_mesh_generic_client_get_state (C++ *function*), 670
esp_ble_mesh_generic_client_get_state_t (C++ *union*), 671
esp_ble_mesh_generic_client_get_state_t::admin_property (C++ *member*), 671
esp_ble_mesh_generic_client_get_state_t::client_property (C++ *member*), 671
esp_ble_mesh_generic_client_get_state_t::manufacturer_property (C++ *member*), 671
esp_ble_mesh_generic_client_get_state_t::user_property (C++ *member*), 671
esp_ble_mesh_generic_client_set_state (C++ *function*), 670
esp_ble_mesh_generic_client_set_state_t (C++ *union*), 671
esp_ble_mesh_generic_client_set_state_t::admin_property (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::def_trans_time (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::delta_set (C++ *member*), 671
esp_ble_mesh_generic_client_set_state_t::level_set (C++ *member*), 671
esp_ble_mesh_generic_client_set_state_t::loc_global (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::loc_local (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::manufacturer_property (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::move_set (C++ *member*), 671
esp_ble_mesh_generic_client_set_state_t::onoff_set (C++ *member*), 671
esp_ble_mesh_generic_client_set_state_t::power_default (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::power_level (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::power_range (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::power_set (C++ *member*), 672
esp_ble_mesh_generic_client_set_state_t::user_property (C++ *member*), 672
esp_ble_mesh_generic_message_opcode_t (C++ *type*), 598
esp_ble_mesh_generic_property_t (C++ *struct*), 693
esp_ble_mesh_generic_property_t::admin_access (C++ *member*), 693
esp_ble_mesh_generic_property_t::id (C++ *member*), 693
esp_ble_mesh_generic_property_t::manu_access

(C++ *member*), 693
esp_ble_mesh_generic_property_t::user_access (C++ *member*), 693
esp_ble_mesh_generic_property_t::val (C++ *member*), 693
esp_ble_mesh_generic_server_cb_event_t (C++ *enum*), 710
esp_ble_mesh_generic_server_cb_event_t::ESP_BLE_MESH_GENERIC_SERVER_EVT_MAX (C++ *enumerator*), 710
esp_ble_mesh_generic_server_cb_event_t::ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT (C++ *enumerator*), 710
esp_ble_mesh_generic_server_cb_event_t::ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT (C++ *enumerator*), 710
esp_ble_mesh_generic_server_cb_event_t::ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT (C++ *enumerator*), 710
esp_ble_mesh_generic_server_cb_param_t (C++ *struct*), 703
esp_ble_mesh_generic_server_cb_param_t::ctx (C++ *member*), 703
esp_ble_mesh_generic_server_cb_param_t::model (C++ *member*), 703
esp_ble_mesh_generic_server_cb_param_t::value (C++ *member*), 703
esp_ble_mesh_generic_server_cb_t (C++ *type*), 709
esp_ble_mesh_generic_server_cb_value_t (C++ *union*), 676
esp_ble_mesh_generic_server_cb_value_t::get (C++ *member*), 676
esp_ble_mesh_generic_server_cb_value_t::set (C++ *member*), 676
esp_ble_mesh_generic_server_recv_get_msg_t (C++ *union*), 675
esp_ble_mesh_generic_server_recv_get_msg_t::admin_property (C++ *member*), 675
esp_ble_mesh_generic_server_recv_get_msg_t::client_properties (C++ *member*), 675
esp_ble_mesh_generic_server_recv_get_msg_t::manu_property (C++ *member*), 675
esp_ble_mesh_generic_server_recv_get_msg_t::user_property (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t (C++ *union*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::admin_property (C++ *member*), 676
esp_ble_mesh_generic_server_recv_set_msg_t::default_trans_time (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::delta (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::level (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::manu_property (C++ *member*), 676
esp_ble_mesh_generic_server_recv_set_msg_t::move (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::onoff (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::onpowerup (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::power (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::tid (C++ *member*), 675
esp_ble_mesh_generic_server_recv_set_msg_t::user_property (C++ *member*), 676
esp_ble_mesh_generic_server_state_change_t (C++ *union*), 674
esp_ble_mesh_generic_server_state_change_t::admin_property (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::def_trans_time (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::delta (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::level (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::loc_global (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::loc_local (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::manu_property (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::move_level (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::onoff (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::onpowerup (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::power (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::power_property (C++ *member*), 674
esp_ble_mesh_generic_server_state_change_t::user_property (C++ *member*), 674
esp_ble_mesh_get_composition_data (C++ *function*), 610
esp_ble_mesh_get_element_count (C++ *function*), 609
esp_ble_mesh_get_fast_prov_app_key (C++ *function*), 620
esp_ble_mesh_get_model_publish_period (C++ *function*), 609
esp_ble_mesh_get_primary_element_address (C++ *function*), 609
ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_COUNT (C *macro*), 577
ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_INTERVAL

(*C macro*), 577
ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT (*C macro*), 727
ESP_BLE_MESH_GET_SENSOR_DATA_LENGTH (*C macro*), 728
ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID (*C macro*), 728
ESP_BLE_MESH_GET_TRANSMIT_COUNT (*C macro*), 576
ESP_BLE_MESH_GET_TRANSMIT_INTERVAL (*C macro*), 577
esp_ble_mesh_health_attention_off_cb_t (*C++ struct*), 666
esp_ble_mesh_health_attention_off_cb_t::model (*C++ member*), 666
esp_ble_mesh_health_attention_on_cb_t (*C++ struct*), 665
esp_ble_mesh_health_attention_on_cb_t::model (*C++ member*), 666
esp_ble_mesh_health_attention_on_cb_t::time (*C++ member*), 666
esp_ble_mesh_health_attention_set_t (*C++ struct*), 663
esp_ble_mesh_health_attention_set_t::attention (*C++ member*), 663
esp_ble_mesh_health_attention_status_cb_t (*C++ struct*), 664
esp_ble_mesh_health_attention_status_cb_t::attention (*C++ member*), 664
esp_ble_mesh_health_client_cb_event_t (*C++ enum*), 669
esp_ble_mesh_health_client_cb_event_t::ESP_BLE_MESH_HEALTH_CLIENT_EVT_MAX (*C++ enumerator*), 669
esp_ble_mesh_health_client_cb_event_t::ESP_BLE_MESH_HEALTH_CLIENT_GET_STATE_EVT (*C++ enumerator*), 669
esp_ble_mesh_health_client_cb_event_t::ESP_BLE_MESH_HEALTH_CLIENT_PUBLISH_EVT (*C++ enumerator*), 669
esp_ble_mesh_health_client_cb_event_t::ESP_BLE_MESH_HEALTH_CLIENT_SET_STATE_EVT (*C++ enumerator*), 669
esp_ble_mesh_health_client_cb_event_t::ESP_BLE_MESH_HEALTH_CLIENT_TIMEOUT_EVT (*C++ enumerator*), 669
esp_ble_mesh_health_client_cb_param_t (*C++ struct*), 664
esp_ble_mesh_health_client_cb_param_t::error_code (*C++ member*), 665
esp_ble_mesh_health_client_cb_param_t::params (*C++ member*), 665
esp_ble_mesh_health_client_cb_param_t::status_cb (*C++ member*), 665
esp_ble_mesh_health_client_cb_t (*C++ type*), 669
esp_ble_mesh_health_client_common_cb_param_t (*C++ union*), 660
esp_ble_mesh_health_client_common_cb_param_t::(anonymous) (*C++ member*), 661
esp_ble_mesh_health_client_common_cb_param_t::ctx (*C++ member*), 660
esp_ble_mesh_health_client_common_cb_param_t::fault_status (*C++ member*), 660
esp_ble_mesh_health_client_common_cb_param_t::opcode (*C++ member*), 660
esp_ble_mesh_health_client_common_cb_param_t::per (*C++ member*), 661
esp_ble_mesh_health_client_get_state (*C++ function*), 659
esp_ble_mesh_health_client_get_state_t (*C++ union*), 660
esp_ble_mesh_health_client_get_state_t::fault_get (*C++ member*), 660
esp_ble_mesh_health_client_set_state (*C++ function*), 659
esp_ble_mesh_health_client_set_state_t (*C++ union*), 660
esp_ble_mesh_health_client_set_state_t::attention (*C++ member*), 660
esp_ble_mesh_health_client_set_state_t::fault_cle (*C++ member*), 660
esp_ble_mesh_health_client_set_state_t::fault_tes (*C++ member*), 660
esp_ble_mesh_health_client_set_state_t::period_se (*C++ member*), 660
esp_ble_mesh_health_current_status_cb_t (*C++ struct*), 663
esp_ble_mesh_health_current_status_cb_t::company_ (*C++ member*), 664
esp_ble_mesh_health_current_status_cb_t::fault_ar (*C++ member*), 664
esp_ble_mesh_health_current_status_cb_t::test_id (*C++ member*), 664
ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE (*C macro*), 669
esp_ble_mesh_health_fault_get_t (*C++ struct*), 662
esp_ble_mesh_health_fault_get_t::company_id (*C++ member*), 663
esp_ble_mesh_health_fault_status_cb_t (*C++ struct*), 664
esp_ble_mesh_health_fault_status_cb_t::company_id (*C++ member*), 664
esp_ble_mesh_health_fault_status_cb_t::fault_arra (*C++ member*), 664
esp_ble_mesh_health_fault_status_cb_t::test_id (*C++ member*), 664
esp_ble_mesh_health_fault_test_cb_t (*C++ struct*), 665
esp_ble_mesh_health_fault_test_cb_t::company_id (*C++ member*), 665
esp_ble_mesh_health_fault_test_cb_t::model (*C++ member*), 665

(C++ *member*), 665
esp_ble_mesh_health_fault_test_cb_t::test_id (C++ *member*), 665
esp_ble_mesh_health_fault_test_t (C++ *struct*), 663
esp_ble_mesh_health_fault_test_t::company_id (C++ *member*), 663
esp_ble_mesh_health_fault_test_t::test_id (C++ *member*), 663
esp_ble_mesh_health_fault_update_comp_data_t (C++ *struct*), 665
esp_ble_mesh_health_fault_update_comp_cb_t::element_addr (C++ *member*), 665
esp_ble_mesh_health_fault_update_comp_cb_t::status (C++ *member*), 665
esp_ble_mesh_health_model_status_t (C++ *type*), 598
esp_ble_mesh_health_period_set_t (C++ *struct*), 663
esp_ble_mesh_health_period_set_t::fast_period_divisor (C++ *member*), 663
esp_ble_mesh_health_period_status_cb_t (C++ *struct*), 664
esp_ble_mesh_health_period_status_cb_t::fast_period_divisor (C++ *member*), 664
ESP_BLE_MESH_HEALTH_PUB_DEFINE (C *macro*), 666
esp_ble_mesh_health_server_cb_event_t (C++ *enum*), 669
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_EVT_MAX (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_event_t::ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT (C++ *enumerator*), 670
esp_ble_mesh_health_server_cb_param_t (C++ *union*), 661
esp_ble_mesh_health_server_cb_param_t::attention_off (C++ *member*), 661
esp_ble_mesh_health_server_cb_param_t::attention_on (C++ *member*), 661
esp_ble_mesh_health_server_cb_param_t::fault_clear (C++ *member*), 661
esp_ble_mesh_health_server_cb_param_t::fault_test (C++ *member*), 661
esp_ble_mesh_health_server_cb_param_t::fault_update_comp (C++ *member*), 661
esp_ble_mesh_health_server_cb_t (C++ *type*), 669
esp_ble_mesh_health_server_fault_update (C++ *function*), 659
esp_ble_mesh_health_srv_cb_t (C++ *struct*), 661
esp_ble_mesh_health_srv_cb_t::attention_off (C++ *member*), 661
esp_ble_mesh_health_srv_cb_t::attention_on (C++ *member*), 661
esp_ble_mesh_health_srv_cb_t::fault_clear (C++ *member*), 661
esp_ble_mesh_health_srv_cb_t::fault_test (C++ *member*), 661
esp_ble_mesh_health_srv_t (C++ *struct*), 662
esp_ble_mesh_health_srv_t::attention_timer (C++ *member*), 662
esp_ble_mesh_health_srv_t::attention_timer_start (C++ *member*), 662
esp_ble_mesh_health_srv_t::health_cb (C++ *member*), 662
esp_ble_mesh_health_srv_t::health_test (C++ *member*), 662
esp_ble_mesh_health_srv_t::model (C++ *member*), 662
ESP_BLE_MESH_HEALTH_STANDARD_TEST (C *macro*), 666
esp_ble_mesh_health_test_t (C++ *struct*), 661
esp_ble_mesh_health_test_t::company_id (C++ *member*), 662
esp_ble_mesh_health_test_t::current_faults (C++ *member*), 662
esp_ble_mesh_health_test_t::id_count (C++ *member*), 662
esp_ble_mesh_health_test_t::prev_test_id (C++ *member*), 662
esp_ble_mesh_health_test_t::registered_faults (C++ *member*), 662
esp_ble_mesh_health_test_t::test_ids (C++ *member*), 662
ESP_BLE_MESH_HEARTBEAT_FILTER_ACCEPTLIST (C *macro*), 579
ESP_BLE_MESH_HEARTBEAT_FILTER_ADD (C *macro*), 579
esp_ble_mesh_heartbeat_filter_info_t (C++ *struct*), 571
esp_ble_mesh_heartbeat_filter_info_t::hb_dst (C++ *member*), 571
esp_ble_mesh_heartbeat_filter_info_t::hb_src (C++ *member*), 571
ESP_BLE_MESH_HEARTBEAT_FILTER_REJECTLIST (C *macro*), 579
ESP_BLE_MESH_HEARTBEAT_FILTER_REMOVE (C *macro*), 579
ESP_BLE_MESH_HOUSING_OPENED_ERROR (C *macro*), 668
ESP_BLE_MESH_HOUSING_OPENED_WARNING (C *macro*), 668
esp_ble_mesh_init (C++ *function*), 608
esp_ble_mesh_input_action_t (C++ *enum*), 599
esp_ble_mesh_input_action_t::ESP_BLE_MESH_ENTER_N

(*C++ enumerator*), 599
esp_ble_mesh_input_action_t::ESP_BLE_MESH_ENTER_STRING (*C++ enumerator*), 599
esp_ble_mesh_input_action_t::ESP_BLE_MESH_NO_INPUT (*C++ enumerator*), 599
esp_ble_mesh_input_action_t::ESP_BLE_MESH_PUSH (*C++ enumerator*), 599
esp_ble_mesh_input_action_t::ESP_BLE_MESH_TWIST (*C++ enumerator*), 599
ESP_BLE_MESH_INPUT_NO_CHANGE_ERROR (*C macro*), 668
ESP_BLE_MESH_INPUT_NO_CHANGE_WARNING (*C macro*), 668
ESP_BLE_MESH_INPUT_TOO_HIGH_ERROR (*C macro*), 668
ESP_BLE_MESH_INPUT_TOO_HIGH_WARNING (*C macro*), 668
ESP_BLE_MESH_INPUT_TOO_LOW_ERROR (*C macro*), 668
ESP_BLE_MESH_INPUT_TOO_LOW_WARNING (*C macro*), 668
ESP_BLE_MESH_INTERNAL_BUS_ERROR (*C macro*), 669
ESP_BLE_MESH_INTERNAL_BUS_WARNING (*C macro*), 669
ESP_BLE_MESH_INVALID_NODE_INDEX (*C macro*), 576
ESP_BLE_MESH_INVALID_SCENE_NUMBER (*C macro*), 756
ESP_BLE_MESH_INVALID_SENSOR_PROPERTY_ID (*C macro*), 726
ESP_BLE_MESH_INVALID_SENSOR_SETTING_PROPERTY_ID (*C macro*), 726
ESP_BLE_MESH_INVALID_SETTINGS_IDX (*C macro*), 574
esp_ble_mesh_is_model_subscribed_to_group (*C++ function*), 609
ESP_BLE_MESH_KEY_ANY (*C macro*), 575
ESP_BLE_MESH_KEY_DEV (*C macro*), 575
ESP_BLE_MESH_KEY_PRIMARY (*C macro*), 575
ESP_BLE_MESH_KEY_UNUSED (*C macro*), 575
ESP_BLE_MESH_KEY_UNUSED_ELT_ (*C macro*), 575
esp_ble_mesh_last_msg_info_t (*C++ struct*), 573
esp_ble_mesh_last_msg_info_t::dst (*C++ member*), 573
esp_ble_mesh_last_msg_info_t::src (*C++ member*), 573
esp_ble_mesh_last_msg_info_t::tid (*C++ member*), 573
esp_ble_mesh_last_msg_info_t::timestamp (*C++ member*), 573
esp_ble_mesh_lc_state_t (*C++ enum*), 813
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_FADE (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_FADE_ON (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_FADE_STA (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_FADE_STA (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_OFF (*C++ enumerator*), 813
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_PROLONG (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_RUN (*C++ enumerator*), 814
esp_ble_mesh_lc_state_t::ESP_BLE_MESH_LC_STANDBY (*C++ enumerator*), 814
esp_ble_mesh_light_client_cb_event_t (*C++ enum*), 813
esp_ble_mesh_light_client_cb_event_t::ESP_BLE_MES (*C++ enumerator*), 813
esp_ble_mesh_light_client_cb_event_t::ESP_BLE_MES (*C++ enumerator*), 813
esp_ble_mesh_light_client_cb_event_t::ESP_BLE_MES (*C++ enumerator*), 813
esp_ble_mesh_light_client_cb_event_t::ESP_BLE_MES (*C++ enumerator*), 813
esp_ble_mesh_light_client_cb_event_t::ESP_BLE_MES (*C++ enumerator*), 813
esp_ble_mesh_light_client_cb_param_t (*C++ struct*), 782
esp_ble_mesh_light_client_cb_param_t::error_code (*C++ member*), 782
esp_ble_mesh_light_client_cb_param_t::params (*C++ member*), 782
esp_ble_mesh_light_client_cb_param_t::status_cb (*C++ member*), 782
esp_ble_mesh_light_client_cb_t (*C++ type*), 813
esp_ble_mesh_light_client_get_state (*C++ function*), 759
esp_ble_mesh_light_client_get_state_t (*C++ union*), 760
esp_ble_mesh_light_client_get_state_t::lc_propert (*C++ member*), 760
esp_ble_mesh_light_client_set_state (*C++ function*), 759
esp_ble_mesh_light_client_set_state_t (*C++ union*), 760
esp_ble_mesh_light_client_set_state_t::ctl_defaul (*C++ member*), 760
esp_ble_mesh_light_client_set_state_t::ctl_set (*C++ member*), 760
esp_ble_mesh_light_client_set_state_t::ctl_temper (*C++ member*), 760
esp_ble_mesh_light_client_set_state_t::ctl_temper (*C++ member*), 760
esp_ble_mesh_light_client_set_state_t::hsl_defaul (*C++ member*), 761
esp_ble_mesh_light_client_set_state_t::hsl_hue_se (*C++ member*), 760
esp_ble_mesh_light_client_set_state_t::hsl_range_ (*C++ member*), 761

esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::lightness_
(C++ member), 761 (C++ member), 762
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::lightness_
(C++ member), 760 (C++ member), 761
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::lightness_
(C++ member), 761 (C++ member), 762
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::lightness_
(C++ member), 761 (C++ member), 761
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::xyl_defaul
(C++ member), 761 (C++ member), 762
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::xyl_range_
(C++ member), 761 (C++ member), 763
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::xyl_status
(C++ member), 760 (C++ member), 762
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_client_status_cb_t::xyl_target
(C++ member), 760 (C++ member), 762
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_control_t (C++ struct),
(C++ member), 760 794
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_control_t::prop_state
(C++ member), 760 (C++ member), 794
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_control_t::state
(C++ member), 761 (C++ member), 794
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_control_t::state_machine
(C++ member), 761 (C++ member), 794
esp_ble_mesh_light_client_set_state_t::esp_ble_mesh_light_ctl_default_set_t
(C++ member), 761 (C++ struct), 769
esp_ble_mesh_light_client_status_cb_t esp_ble_mesh_light_ctl_default_set_t::delta_uv
(C++ union), 761 (C++ member), 769
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_set_t::lightness
(C++ member), 762 (C++ member), 769
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_set_t::temperature
(C++ member), 762 (C++ member), 769
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_status_cb_t
(C++ member), 762 (C++ struct), 777
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_status_cb_t::delta
(C++ member), 762 (C++ member), 777
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_status_cb_t::light
(C++ member), 762 (C++ member), 777
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_default_status_cb_t::tempe
(C++ member), 762 (C++ member), 777
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t (C++ struct),
(C++ member), 762 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::ctl_delta_uv
(C++ member), 762 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::ctl_lightness
(C++ member), 762 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::ctl_temperatrue
(C++ member), 762 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::delay
(C++ member), 763 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::op_en
(C++ member), 763 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::tid
(C++ member), 763 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_ctl_set_t::trans_time
(C++ member), 763 (C++ member), 768
esp_ble_mesh_light_client_status_cb_t::esp_ble_mesh_light_setup_srv_t
(C++ member), 762 (C++ struct), 785

esp_ble_mesh_light_ctl_setup_srv_t::model (C++ member), 785
esp_ble_mesh_light_ctl_setup_srv_t::rsp_ctrl (C++ member), 785
esp_ble_mesh_light_ctl_setup_srv_t::state (C++ member), 785
esp_ble_mesh_light_ctl_srv_t (C++ struct), 784
esp_ble_mesh_light_ctl_srv_t::last (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::model (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::rsp_ctrl (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::state (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::transition (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::tt_delta_delta_uv (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::tt_delta_lightness (C++ member), 785
esp_ble_mesh_light_ctl_srv_t::tt_delta_temperature (C++ member), 785
esp_ble_mesh_light_ctl_state_t (C++ struct), 784
esp_ble_mesh_light_ctl_state_t::delta_uv (C++ member), 784
esp_ble_mesh_light_ctl_state_t::delta_uv_default (C++ member), 784
esp_ble_mesh_light_ctl_state_t::lightness (C++ member), 784
esp_ble_mesh_light_ctl_state_t::lightness_default (C++ member), 784
esp_ble_mesh_light_ctl_state_t::status_code (C++ member), 784
esp_ble_mesh_light_ctl_state_t::target_delta_uv (C++ member), 784
esp_ble_mesh_light_ctl_state_t::target_lightness (C++ member), 784
esp_ble_mesh_light_ctl_state_t::target_temperature (C++ member), 784
esp_ble_mesh_light_ctl_state_t::temperature (C++ member), 784
esp_ble_mesh_light_ctl_state_t::temperature_default (C++ member), 784
esp_ble_mesh_light_ctl_state_t::temperature_range_max (C++ member), 784
esp_ble_mesh_light_ctl_state_t::temperature_range_min (C++ member), 784
esp_ble_mesh_light_ctl_status_cb_t (C++ struct), 775
esp_ble_mesh_light_ctl_status_cb_t::op_en (C++ member), 775
esp_ble_mesh_light_ctl_status_cb_t::present_ctl_lightness (C++ member), 775
esp_ble_mesh_light_ctl_status_cb_t::present_ctl_temperature (C++ member), 775
esp_ble_mesh_light_ctl_status_cb_t::remain_time (C++ member), 776
esp_ble_mesh_light_ctl_status_cb_t::target_ctl_lightness (C++ member), 776
esp_ble_mesh_light_ctl_status_cb_t::target_ctl_temperature (C++ member), 776
esp_ble_mesh_light_ctl_temp_srv_t (C++ struct), 785
esp_ble_mesh_light_ctl_temp_srv_t::last (C++ member), 786
esp_ble_mesh_light_ctl_temp_srv_t::model (C++ member), 785
esp_ble_mesh_light_ctl_temp_srv_t::rsp_ctrl (C++ member), 785
esp_ble_mesh_light_ctl_temp_srv_t::state (C++ member), 786
esp_ble_mesh_light_ctl_temp_srv_t::transition (C++ member), 786
esp_ble_mesh_light_ctl_temp_srv_t::tt_delta_delta_uv (C++ member), 786
esp_ble_mesh_light_ctl_temp_srv_t::tt_delta_temperature (C++ member), 786
esp_ble_mesh_light_ctl_temperature_range_set_t (C++ struct), 769
esp_ble_mesh_light_ctl_temperature_range_set_t::range_max (C++ member), 769
esp_ble_mesh_light_ctl_temperature_range_set_t::range_min (C++ member), 769
esp_ble_mesh_light_ctl_temperature_range_status_cb_t (C++ struct), 776
esp_ble_mesh_light_ctl_temperature_range_status_cb_t::range_max (C++ member), 776
esp_ble_mesh_light_ctl_temperature_range_status_cb_t::range_min (C++ member), 776
esp_ble_mesh_light_ctl_temperature_range_status_cb_t::status_code (C++ member), 776
esp_ble_mesh_light_ctl_temperature_set_t (C++ struct), 768
esp_ble_mesh_light_ctl_temperature_set_t::ctl_delta_uv (C++ member), 768
esp_ble_mesh_light_ctl_temperature_set_t::ctl_temperature (C++ member), 768
esp_ble_mesh_light_ctl_temperature_set_t::delay (C++ member), 769
esp_ble_mesh_light_ctl_temperature_set_t::op_en (C++ member), 768
esp_ble_mesh_light_ctl_temperature_set_t::tid (C++ member), 768
esp_ble_mesh_light_ctl_temperature_set_t::trans_time (C++ member), 769
esp_ble_mesh_light_ctl_temperature_status_cb_t (C++ struct), 776
esp_ble_mesh_light_ctl_temperature_status_cb_t::op_en (C++ member), 776
esp_ble_mesh_light_ctl_temperature_status_cb_t::present_ctl_delta_uv (C++ member), 776
esp_ble_mesh_light_ctl_temperature_status_cb_t::present_ctl_temperature (C++ member), 776

esp_ble_mesh_light_ctl_temperature_sta (C++ member), 776
esp_ble_mesh_light_ctl_temperature_sta (C++ member), 776
esp_ble_mesh_light_ctl_temperature_sta (C++ member), 776
esp_ble_mesh_light_hsl_default_set_t (C++ struct), 771
esp_ble_mesh_light_hsl_default_set_t::ha (C++ member), 771
esp_ble_mesh_light_hsl_default_set_t::le (C++ member), 771
esp_ble_mesh_light_hsl_default_set_t::sa (C++ member), 771
esp_ble_mesh_light_hsl_default_status_cb (C++ struct), 778
esp_ble_mesh_light_hsl_default_status_cb (C++ member), 779
esp_ble_mesh_light_hsl_default_status_cb (C++ member), 779
esp_ble_mesh_light_hsl_default_status_cb (C++ member), 779
esp_ble_mesh_light_hsl_hue_set_t (C++ struct), 770
esp_ble_mesh_light_hsl_hue_set_t::delay (C++ member), 770
esp_ble_mesh_light_hsl_hue_set_t::hue (C++ member), 770
esp_ble_mesh_light_hsl_hue_set_t::op_en (C++ member), 770
esp_ble_mesh_light_hsl_hue_set_t::tid (C++ member), 770
esp_ble_mesh_light_hsl_hue_set_t::trans (C++ member), 770
esp_ble_mesh_light_hsl_hue_srv_t (C++ struct), 788
esp_ble_mesh_light_hsl_hue_srv_t::last (C++ member), 788
esp_ble_mesh_light_hsl_hue_srv_t::mode (C++ member), 788
esp_ble_mesh_light_hsl_hue_srv_t::rsp_c (C++ member), 788
esp_ble_mesh_light_hsl_hue_srv_t::state (C++ member), 788
esp_ble_mesh_light_hsl_hue_srv_t::trans (C++ member), 788
esp_ble_mesh_light_hsl_hue_srv_t::tt_de (C++ member), 788
esp_ble_mesh_light_hsl_hue_status_cb_t (C++ struct), 778
esp_ble_mesh_light_hsl_hue_status_cb_t: (C++ member), 778
esp_ble_mesh_light_hsl_hue_status_cb_t: (C++ member), 778
esp_ble_mesh_light_hsl_hue_status_cb_t: (C++ member), 778
esp_ble_mesh_light_hsl_hue_status_cb_t: (C++ member), 778
esp_ble_mesh_light_hsl_range_set_t (C++ struct), 771
esp_ble_mesh_light_hsl_range_set_t::hue (C++ member), 771
esp_ble_mesh_light_hsl_range_set_t::hue (C++ member), 771
esp_ble_mesh_light_hsl_range_set_t::op_e (C++ member), 771
esp_ble_mesh_light_hsl_range_set_t::satu (C++ member), 771
esp_ble_mesh_light_hsl_range_set_t::satu (C++ member), 771
esp_ble_mesh_light_hsl_range_status_cb_t (C++ struct), 779
esp_ble_mesh_light_hsl_range_status_cb_t::hue_ran (C++ member), 779
esp_ble_mesh_light_hsl_range_status_cb_t::hue_ran (C++ member), 779
esp_ble_mesh_light_hsl_range_status_cb_t::saturat (C++ member), 779
esp_ble_mesh_light_hsl_range_status_cb_t::saturat (C++ member), 779
esp_ble_mesh_light_hsl_range_status_cb_t::status_ (C++ member), 779
esp_ble_mesh_light_hsl_sat_srv_t (C++ struct), 788
esp_ble_mesh_light_hsl_sat_srv_t::last (C++ member), 788
esp_ble_mesh_light_hsl_sat_srv_t::model (C++ member), 788
esp_ble_mesh_light_hsl_sat_srv_t::rsp_ctrl (C++ member), 788
esp_ble_mesh_light_hsl_sat_srv_t::state (C++ member), 788
esp_ble_mesh_light_hsl_sat_srv_t::transition (C++ member), 789
esp_ble_mesh_light_hsl_sat_srv_t::tt_delta_satura (C++ member), 789
esp_ble_mesh_light_hsl_saturation_set_t (C++ struct), 770
esp_ble_mesh_light_hsl_saturation_set_t::delay (C++ member), 771
esp_ble_mesh_light_hsl_saturation_set_t::op_en (C++ member), 770
esp_ble_mesh_light_hsl_saturation_set_t::saturati (C++ member), 770
esp_ble_mesh_light_hsl_saturation_set_t::tid (C++ member), 770
esp_ble_mesh_light_hsl_saturation_set_t::trans_ti (C++ member), 771
esp_ble_mesh_light_hsl_saturation_status_cb_t (C++ struct), 778
esp_ble_mesh_light_hsl_saturation_status_cb_t::op (C++ member), 778
esp_ble_mesh_light_hsl_saturation_status_cb_t::pr (C++ member), 778
esp_ble_mesh_light_hsl_saturation_status_cb_t::re (C++ member), 778
esp_ble_mesh_light_hsl_saturation_status_cb_t::ta (C++ member), 778

esp_ble_mesh_light_hsl_set_t (*C++ struct*), 769
esp_ble_mesh_light_hsl_set_t::delay (*C++ member*), 770
esp_ble_mesh_light_hsl_set_t::hsl_hue (*C++ member*), 769
esp_ble_mesh_light_hsl_set_t::hsl_lightness (*C++ member*), 769
esp_ble_mesh_light_hsl_set_t::hsl_saturation (*C++ member*), 770
esp_ble_mesh_light_hsl_set_t::op_en (*C++ member*), 769
esp_ble_mesh_light_hsl_set_t::tid (*C++ member*), 770
esp_ble_mesh_light_hsl_set_t::trans_time (*C++ member*), 770
esp_ble_mesh_light_hsl_setup_srv_t (*C++ struct*), 787
esp_ble_mesh_light_hsl_setup_srv_t::model (*C++ member*), 788
esp_ble_mesh_light_hsl_setup_srv_t::rsp_ctrl (*C++ member*), 788
esp_ble_mesh_light_hsl_setup_srv_t::state (*C++ member*), 788
esp_ble_mesh_light_hsl_srv_t (*C++ struct*), 787
esp_ble_mesh_light_hsl_srv_t::last (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::model (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::rsp_ctrl (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::state (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::transition (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::tt_delta_hue (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::tt_delta_lightness (*C++ member*), 787
esp_ble_mesh_light_hsl_srv_t::tt_delta_saturation (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t (*C++ struct*), 786
esp_ble_mesh_light_hsl_state_t::hue (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::hue_default (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::hue_range_max (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t::hue_range_min (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t::lightness (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::lightness_default (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::saturation (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::saturation_default (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::saturation_range_max (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t::saturation_range_min (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t::status_code (*C++ member*), 787
esp_ble_mesh_light_hsl_state_t::target_hue (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::target_lightness (*C++ member*), 786
esp_ble_mesh_light_hsl_state_t::target_saturation (*C++ member*), 786
esp_ble_mesh_light_hsl_status_cb_t (*C++ struct*), 777
esp_ble_mesh_light_hsl_status_cb_t::hsl_hue (*C++ member*), 777
esp_ble_mesh_light_hsl_status_cb_t::hsl_lightness (*C++ member*), 777
esp_ble_mesh_light_hsl_status_cb_t::hsl_saturation (*C++ member*), 777
esp_ble_mesh_light_hsl_status_cb_t::op_en (*C++ member*), 777
esp_ble_mesh_light_hsl_status_cb_t::remain_time (*C++ member*), 777
esp_ble_mesh_light_hsl_target_status_cb_t (*C++ struct*), 777
esp_ble_mesh_light_hsl_target_status_cb_t::hsl_hue (*C++ member*), 778
esp_ble_mesh_light_hsl_target_status_cb_t::hsl_lightness (*C++ member*), 777
esp_ble_mesh_light_hsl_target_status_cb_t::hsl_saturation (*C++ member*), 778
esp_ble_mesh_light_hsl_target_status_cb_t::op_en (*C++ member*), 777
esp_ble_mesh_light_hsl_target_status_cb_t::remain_time (*C++ member*), 778
esp_ble_mesh_light_lc_light_onoff_set_t (*C++ struct*), 773
esp_ble_mesh_light_lc_light_onoff_set_t::delay (*C++ member*), 773
esp_ble_mesh_light_lc_light_onoff_set_t::light_onoff (*C++ member*), 773
esp_ble_mesh_light_lc_light_onoff_set_t::op_en (*C++ member*), 773
esp_ble_mesh_light_lc_light_onoff_set_t::tid (*C++ member*), 773
esp_ble_mesh_light_lc_light_onoff_set_t::trans_time (*C++ member*), 773
esp_ble_mesh_light_lc_light_onoff_status_cb_t (*C++ struct*), 781
esp_ble_mesh_light_lc_light_onoff_status_cb_t::op_en (*C++ member*), 781
esp_ble_mesh_light_lc_light_onoff_status_cb_t::present_onoff (*C++ member*), 781
esp_ble_mesh_light_lc_light_onoff_status_cb_t::remain_time (*C++ member*), 781

esp_ble_mesh_light_lc_light_onoff_status_cb_t::esp_ble_mesh_light_lc_property_state_t::time_fade
(C++ member), 781 (C++ member), 792
esp_ble_mesh_light_lc_mode_set_t (C++ esp_ble_mesh_light_lc_property_state_t::time_fade
struct), 773 (C++ member), 792
esp_ble_mesh_light_lc_mode_set_t::mode esp_ble_mesh_light_lc_property_state_t::time_occu
(C++ member), 773 (C++ member), 791
esp_ble_mesh_light_lc_mode_status_cb_t esp_ble_mesh_light_lc_property_state_t::time_prol
(C++ struct), 781 (C++ member), 792
esp_ble_mesh_light_lc_mode_status_cb_t::esp_ble_mesh_light_lc_property_state_t::time_run_
(C++ member), 781 (C++ member), 792
esp_ble_mesh_light_lc_om_set_t (C++ esp_ble_mesh_light_lc_property_status_cb_t
struct), 773 (C++ struct), 781
esp_ble_mesh_light_lc_om_set_t::mode esp_ble_mesh_light_lc_property_status_cb_t::prope
(C++ member), 773 (C++ member), 782
esp_ble_mesh_light_lc_om_status_cb_t esp_ble_mesh_light_lc_property_status_cb_t::prope
(C++ struct), 781 (C++ member), 782
esp_ble_mesh_light_lc_om_status_cb_t::mode esp_ble_mesh_light_lc_setup_srv_t
(C++ member), 781 (C++ struct), 795
esp_ble_mesh_light_lc_property_get_t esp_ble_mesh_light_lc_setup_srv_t::lc
(C++ struct), 773 (C++ member), 795
esp_ble_mesh_light_lc_property_get_t::esp_ble_mesh_light_lc_setup_srv_t::model
(C++ member), 774 (C++ member), 795
esp_ble_mesh_light_lc_property_set_t esp_ble_mesh_light_lc_setup_srv_t::rsp_ctrl
(C++ struct), 774 (C++ member), 795
esp_ble_mesh_light_lc_property_set_t::esp_ble_mesh_light_lc_srv_t (C++ struct),
(C++ member), 774 794
esp_ble_mesh_light_lc_property_set_t::esp_ble_mesh_light_lc_srv_t::last
(C++ member), 774 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t esp_ble_mesh_light_lc_srv_t::lc (C++
(C++ struct), 791 member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_srv_t::model
(C++ member), 792 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_srv_t::rsp_ctrl
(C++ member), 792 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_srv_t::transition
(C++ member), 792 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t
(C++ member), 792 (C++ struct), 793
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::fade
(C++ member), 792 (C++ member), 793
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::fade_on
(C++ member), 792 (C++ member), 793
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::fade_stand
(C++ member), 793 (C++ member), 793
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::fade_stand
(C++ member), 793 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::state
(C++ member), 793 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::timer
(C++ member), 793 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_machine_t::trans_time
(C++ member), 793 (C++ member), 794
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_t (C++
(C++ member), 793 struct), 791
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_t::ambient_luxlevel
(C++ member), 792 (C++ member), 791
esp_ble_mesh_light_lc_property_state_t::esp_ble_mesh_light_lc_state_t::light_onoff
(C++ member), 792 (C++ member), 791

esp_ble_mesh_light_lc_state_t::linear_output (C++ member), 791
esp_ble_mesh_light_lc_state_t::mode (C++ member), 791
esp_ble_mesh_light_lc_state_t::occupancy (C++ member), 791
esp_ble_mesh_light_lc_state_t::occupancy_mode (C++ member), 791
esp_ble_mesh_light_lc_state_t::target_light_onoff (C++ member), 791
esp_ble_mesh_light_lightness_default_set_t (C++ struct), 767
esp_ble_mesh_light_lightness_default_set_t::lightness (C++ member), 767
esp_ble_mesh_light_lightness_default_status_cb (C++ struct), 775
esp_ble_mesh_light_lightness_default_status_cb::lightness (C++ member), 775
esp_ble_mesh_light_lightness_last_status_cb (C++ struct), 775
esp_ble_mesh_light_lightness_last_status_cb::lightness (C++ member), 775
esp_ble_mesh_light_lightness_linear_set_t (C++ struct), 767
esp_ble_mesh_light_lightness_linear_set_t::delay (C++ member), 767
esp_ble_mesh_light_lightness_linear_set_t::lightness (C++ member), 767
esp_ble_mesh_light_lightness_linear_set_t::op_en (C++ member), 767
esp_ble_mesh_light_lightness_linear_set_t::tid (C++ member), 767
esp_ble_mesh_light_lightness_linear_set_t::trans_time (C++ member), 767
esp_ble_mesh_light_lightness_linear_status_cb (C++ struct), 774
esp_ble_mesh_light_lightness_linear_status_cb::lightness_linear (C++ member), 774
esp_ble_mesh_light_lightness_linear_status_cb::lightness_target (C++ member), 774
esp_ble_mesh_light_lightness_linear_status_cb::remain_time (C++ member), 775
esp_ble_mesh_light_lightness_linear_status_cb::target_lightness (C++ member), 774
esp_ble_mesh_light_lightness_range_set_t (C++ struct), 767
esp_ble_mesh_light_lightness_range_set_t::range_max (C++ member), 768
esp_ble_mesh_light_lightness_range_set_t::range_min (C++ member), 768
esp_ble_mesh_light_lightness_range_status_cb (C++ struct), 775
esp_ble_mesh_light_lightness_range_status_cb::max_range (C++ member), 775
esp_ble_mesh_light_lightness_range_status_cb::min_range (C++ member), 775
esp_ble_mesh_light_lightness_range_status_cb::status_code (C++ member), 775
esp_ble_mesh_light_lightness_set_t (C++ struct), 766
esp_ble_mesh_light_lightness_set_t::delay (C++ member), 767
esp_ble_mesh_light_lightness_set_t::lightness (C++ member), 767
esp_ble_mesh_light_lightness_set_t::op_en (C++ member), 766
esp_ble_mesh_light_lightness_set_t::tid (C++ member), 767
esp_ble_mesh_light_lightness_set_t::trans_time (C++ member), 767
esp_ble_mesh_light_lightness_setup_srv_t (C++ struct), 783
esp_ble_mesh_light_lightness_setup_srv_t::model (C++ member), 783
esp_ble_mesh_light_lightness_setup_srv_t::rsp_ctrl (C++ member), 783
esp_ble_mesh_light_lightness_setup_srv_t::state (C++ member), 783
esp_ble_mesh_light_lightness_srv_t (C++ struct), 783
esp_ble_mesh_light_lightness_srv_t::actual_transi (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::last (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::linear_transi (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::model (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::rsp_ctrl (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::state (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::tt_delta_ligh (C++ member), 783
esp_ble_mesh_light_lightness_srv_t::tt_delta_ligh (C++ member), 783
esp_ble_mesh_light_lightness_state_t (C++ struct), 782
esp_ble_mesh_light_lightness_state_t::lightness_a (C++ member), 782
esp_ble_mesh_light_lightness_state_t::lightness_d (C++ member), 782
esp_ble_mesh_light_lightness_state_t::lightness_l (C++ member), 782
esp_ble_mesh_light_lightness_state_t::lightness_l (C++ member), 782
esp_ble_mesh_light_lightness_state_t::lightness_r (C++ member), 783
esp_ble_mesh_light_lightness_state_t::lightness_r (C++ member), 782
esp_ble_mesh_light_lightness_state_t::status_code (C++ member), 782
esp_ble_mesh_light_lightness_state_t::target_ligh (C++ member), 782
esp_ble_mesh_light_lightness_state_t::target_ligh (C++ member), 782

esp_ble_mesh_light_lightness_status_cb_t (C++ struct), 774
esp_ble_mesh_light_lightness_status_cb_t::current (C++ member), 774
esp_ble_mesh_light_lightness_status_cb_t::remain_time (C++ member), 774
esp_ble_mesh_light_lightness_status_cb_t::target (C++ member), 774
esp_ble_mesh_light_message_opcode_t (C++ type), 598
esp_ble_mesh_light_xyl_default_set_t (C++ struct), 772
esp_ble_mesh_light_xyl_default_set_t::lightness (C++ member), 772
esp_ble_mesh_light_xyl_default_set_t::xyl_x (C++ member), 772
esp_ble_mesh_light_xyl_default_set_t::xyl_y (C++ member), 772
esp_ble_mesh_light_xyl_default_status_cb_t (C++ struct), 780
esp_ble_mesh_light_xyl_default_status_cb_t::lightness (C++ member), 780
esp_ble_mesh_light_xyl_default_status_cb_t::xyl_x (C++ member), 780
esp_ble_mesh_light_xyl_default_status_cb_t::xyl_y (C++ member), 780
esp_ble_mesh_light_xyl_range_set_t (C++ struct), 772
esp_ble_mesh_light_xyl_range_set_t::xyl_range_max (C++ member), 772
esp_ble_mesh_light_xyl_range_set_t::xyl_range_min (C++ member), 772
esp_ble_mesh_light_xyl_range_set_t::xyl_range_max (C++ member), 773
esp_ble_mesh_light_xyl_range_set_t::xyl_range_min (C++ member), 772
esp_ble_mesh_light_xyl_range_status_cb_t (C++ struct), 780
esp_ble_mesh_light_xyl_range_status_cb_t::status_code (C++ member), 780
esp_ble_mesh_light_xyl_range_status_cb_t::xyl_x_range_max (C++ member), 781
esp_ble_mesh_light_xyl_range_status_cb_t::xyl_x_range_min (C++ member), 780
esp_ble_mesh_light_xyl_range_status_cb_t::xyl_y_range_max (C++ member), 781
esp_ble_mesh_light_xyl_range_status_cb_t::xyl_y_range_min (C++ member), 781
esp_ble_mesh_light_xyl_set_t (C++ struct), 771
esp_ble_mesh_light_xyl_set_t::delay (C++ member), 772
esp_ble_mesh_light_xyl_set_t::op_en (C++ member), 771
esp_ble_mesh_light_xyl_set_t::tid (C++ member), 772
esp_ble_mesh_light_xyl_set_t::trans_time (C++ member), 772
esp_ble_mesh_light_xyl_set_t::xyl_lightness (C++ member), 772
esp_ble_mesh_light_xyl_set_t::xyl_x (C++ member), 772
esp_ble_mesh_light_xyl_set_t::xyl_y (C++ member), 772
esp_ble_mesh_light_xyl_setup_srv_t (C++ struct), 790
esp_ble_mesh_light_xyl_setup_srv_t::model (C++ member), 790
esp_ble_mesh_light_xyl_setup_srv_t::rsp_ctrl (C++ member), 790
esp_ble_mesh_light_xyl_setup_srv_t::state (C++ member), 791
esp_ble_mesh_light_xyl_srv_t (C++ struct), 790
esp_ble_mesh_light_xyl_srv_t::last (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::model (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::rsp_ctrl (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::state (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::transition (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::tt_delta_lightness (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::tt_delta_x (C++ member), 790
esp_ble_mesh_light_xyl_srv_t::tt_delta_y (C++ member), 790
esp_ble_mesh_light_xyl_state_t (C++ struct), 789
esp_ble_mesh_light_xyl_state_t::lightness (C++ member), 789
esp_ble_mesh_light_xyl_state_t::lightness_default (C++ member), 789
esp_ble_mesh_light_xyl_state_t::status_code (C++ member), 789
esp_ble_mesh_light_xyl_state_t::target_lightness (C++ member), 789
esp_ble_mesh_light_xyl_state_t::target_x (C++ member), 789
esp_ble_mesh_light_xyl_state_t::target_y (C++ member), 789
esp_ble_mesh_light_xyl_state_t::x (C++ member), 789
esp_ble_mesh_light_xyl_state_t::x_default (C++ member), 789
esp_ble_mesh_light_xyl_state_t::x_range_max (C++ member), 789
esp_ble_mesh_light_xyl_state_t::x_range_min (C++ member), 789
esp_ble_mesh_light_xyl_state_t::y (C++ member), 789

esp_ble_mesh_light_xyl_state_t::y_defau (C++ member), 789
esp_ble_mesh_light_xyl_state_t::y_range (C++ member), 790
esp_ble_mesh_light_xyl_state_t::y_range (C++ member), 790
esp_ble_mesh_light_xyl_status_cb_t (C++ struct), 779
esp_ble_mesh_light_xyl_status_cb_t::op_ (C++ member), 779
esp_ble_mesh_light_xyl_status_cb_t::rem (C++ member), 779
esp_ble_mesh_light_xyl_status_cb_t::xyl (C++ member), 779
esp_ble_mesh_light_xyl_status_cb_t::xyl (C++ member), 779
esp_ble_mesh_light_xyl_status_cb_t::xyl (C++ member), 779
esp_ble_mesh_light_xyl_target_status_cb (C++ struct), 780
esp_ble_mesh_light_xyl_target_status_cb (C++ member), 780
esp_ble_mesh_light_xyl_target_status_cb (C++ member), 780
esp_ble_mesh_light_xyl_target_status_cb (C++ member), 780
esp_ble_mesh_light_xyl_target_status_cb (C++ member), 780
esp_ble_mesh_lighting_server_cb_event_t (C++ enum), 814
esp_ble_mesh_lighting_server_cb_event_t (C++ enumerator), 814
esp_ble_mesh_lighting_server_cb_event_t (C++ enumerator), 814
esp_ble_mesh_lighting_server_cb_event_t (C++ enumerator), 814
esp_ble_mesh_lighting_server_cb_event_t (C++ enumerator), 814
esp_ble_mesh_lighting_server_cb_event_t (C++ enumerator), 814
esp_ble_mesh_lighting_server_cb_param_t (C++ struct), 808
esp_ble_mesh_lighting_server_cb_param_t (C++ member), 808
esp_ble_mesh_lighting_server_cb_param_t (C++ member), 808
esp_ble_mesh_lighting_server_cb_param_t (C++ member), 808
esp_ble_mesh_lighting_server_cb_t (C++ type), 813
esp_ble_mesh_lighting_server_cb_value_t (C++ union), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_cb_value_t (C++ member), 766
esp_ble_mesh_lighting_server_recv_get_msg_t (C++ union), 764
esp_ble_mesh_lighting_server_recv_get_msg_t (C++ member), 764
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ union), 764
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 765
esp_ble_mesh_lighting_server_recv_set_msg_t (C++ member), 766
esp_ble_mesh_lighting_server_recv_status_msg_t (C++ union), 766
esp_ble_mesh_lighting_server_recv_status_msg_t (C++ member), 766
esp_ble_mesh_lighting_server_state_change_t (C++ union), 763
esp_ble_mesh_lighting_server_state_change_t (C++ member), 763

esp_ble_mesh_lighting_server_state_change_t::ber), 566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::member), 566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::ber), 566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::member), 566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 763
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lighting_server_state_change_t::566
(C++ member), 764
esp_ble_mesh_lpn_disable (C++ function), 611
esp_ble_mesh_lpn_enable (C++ function), 611
esp_ble_mesh_lpn_poll (C++ function), 611
ESP_BLE_MESH_MECHANISM_JAMMED_ERROR (C macro), 669
ESP_BLE_MESH_MECHANISM_JAMMED_WARNING (C macro), 669
ESP_BLE_MESH_MEMORY_ERROR (C macro), 668
ESP_BLE_MESH_MEMORY_WARNING (C macro), 668
ESP_BLE_MESH_MIC_LONG (C macro), 574
ESP_BLE_MESH_MIC_SHORT (C macro), 574
esp_ble_mesh_model (C++ struct), 565
esp_ble_mesh_model::cb (C++ member), 566
esp_ble_mesh_model::company_id (C++ member), 566
esp_ble_mesh_model::element (C++ member), 566
esp_ble_mesh_model::element_idx (C++ member), 566
esp_ble_mesh_model::flags (C++ member), 566
esp_ble_mesh_model::groups (C++ member), 566
esp_ble_mesh_model::keys (C++ member), 566
esp_ble_mesh_model::model_id (C++ member), 566
esp_ble_mesh_model::model_idx (C++ member), 566
esp_ble_mesh_model::op (C++ member), 566
esp_ble_mesh_model::pub (C++ member), 566
esp_ble_mesh_model::user_data (C++ member), 566
esp_ble_mesh_model::vnd (C++ member), 566
esp_ble_mesh_model::[anonymous] (C++ member), 566
esp_ble_mesh_model_cb_event_t (C++ enum), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_CLIEN (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_CLIEN (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_MODEL (C++ enumerator), 608
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_MODEL (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_MODEL (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_MODEL (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_MODEL (C++ enumerator), 607
esp_ble_mesh_model_cb_event_t::ESP_BLE_MESH_SERVE (C++ enumerator), 608
esp_ble_mesh_model_cb_param_t (C++ union), 560
esp_ble_mesh_model_cb_param_t::ble_mesh_client_mo (C++ struct), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_client_mo (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_client_mo (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_client_mo (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_mod_recv_ (C++ struct), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_mod_recv_ (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_mod_recv_ (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_mod_recv_ (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_mod_recv_ (C++ member), 561

esp_ble_mesh_model_cb_param_t::ble_mesh_model_publish_comp_param::opcode (C++ member), 565
esp_ble_mesh_model_cb_param_t::ble_mesh_model_send_comp_param::err_code (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_model_send_comp_param::opcode (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_model_send_comp_param::value (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_model_send_comp_param::value_len (C++ member), 561
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param (C++ struct), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param::ctx (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param::model (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param::type (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param::value (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_state_change_param::value_len (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_update_appkey_comp_param (C++ struct), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_update_appkey_comp_param::err_code (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_update_appkey_comp_param::model_handle (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_model_update_appkey_comp_param::net_idx (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_server_model_update_state_comp_param (C++ struct), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_server_model_update_state_comp_param::err_code (C++ member), 562
esp_ble_mesh_model_cb_param_t::ble_mesh_server_model_update_state_comp_param::type (C++ member), 562
esp_ble_mesh_model_cb_param_t::client_recv_publish_msg_param (C++ member), 560
esp_ble_mesh_model_cb_param_t::client_send_timeout_param (C++ member), 561
esp_ble_mesh_model_cb_param_t::model_operation_param (C++ member), 560
esp_ble_mesh_model_cb_param_t::model_publish_param (C++ member), 560
esp_ble_mesh_model_cb_param_t::model_publish_update_param (C++ member), 561
esp_ble_mesh_model_cb_param_t::model_send_comp_param (C++ member), 560
esp_ble_mesh_model_cb_param_t::server_model_update_state_param (C++ member), 561
esp_ble_mesh_model_cb_t (C++ type), 620
esp_ble_mesh_model_cbs_t (C++ struct), 565
ESP_BLE_MESH_MODEL_CFG_CLI (C macro), 658
ESP_BLE_MESH_MODEL_CFG_SRV (C macro), 657
ESP_BLE_MESH_MODEL_GEN_ADMIN_PROP_SRV (C macro), 708
ESP_BLE_MESH_MODEL_GEN_BATTERY_CLI (C macro), 704
ESP_BLE_MESH_MODEL_GEN_BATTERY_SRV (C macro), 707
ESP_BLE_MESH_MODEL_GEN_CLIENT_PROP_SRV (C macro), 708
ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_CLI (C macro), 704
ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_SRV (C macro), 705
ESP_BLE_MESH_MODEL_GEN_LEVEL_CLI (C macro), 703
ESP_BLE_MESH_MODEL_GEN_LEVEL_SRV (C macro), 705
ESP_BLE_MESH_MODEL_GEN_LOCATION_CLI (C macro), 705
ESP_BLE_MESH_MODEL_GEN_LOCATION_SETUP_SRV (C macro), 707
ESP_BLE_MESH_MODEL_GEN_LOCATION_SRV (C macro), 707
ESP_BLE_MESH_MODEL_GEN_MANUFACTURER_PROP_SRV (C macro), 708
ESP_BLE_MESH_MODEL_GEN_ONOFF_CLI (C macro), 703
ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV (C macro), 705
ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_CLI (C macro), 704
ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SETUP_SRV (C macro), 706
ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SRV (C macro), 706
ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_CLI (C macro), 704
ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SETUP_SRV (C macro), 706
ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SRV (C macro), 706
ESP_BLE_MESH_MODEL_GEN_PROPERTY_CLI (C macro), 705
ESP_BLE_MESH_MODEL_GEN_USER_PROP_SRV (C macro), 707
ESP_BLE_MESH_MODEL_GROUPS_UNASSIGNED (C macro), 575
ESP_BLE_MESH_MODEL_HEALTH_CLI (C macro), 666
ESP_BLE_MESH_MODEL_HEALTH_SRV (C macro), 666
ESP_BLE_MESH_MODEL_ID_CONFIG_CLI (C macro), 579
ESP_BLE_MESH_MODEL_ID_CONFIG_SRV (C macro), 579

ESP_BLE_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_CLI (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_SRV (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SETUP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_CLI (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_SRV (C macro), 579
ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_PROP_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_GEN_USER_PROP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_HEALTH_CLI (C macro), 579
ESP_BLE_MESH_MODEL_ID_HEALTH_SRV (C macro), 579
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_SCENE_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_SCENE_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_SCENE_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_SCHEDULER_CLI (C macro), 581
ESP_BLE_MESH_MODEL_ID_SCHEDULER_SETUP_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_SCHEDULER_SRV (C macro), 581
ESP_BLE_MESH_MODEL_ID_SENSOR_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_SENSOR_SETUP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_SENSOR_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_TIME_CLI (C macro), 580
ESP_BLE_MESH_MODEL_ID_TIME_SETUP_SRV (C macro), 580
ESP_BLE_MESH_MODEL_ID_TIME_SRV (C macro), 580
ESP_BLE_MESH_MODEL_KEYS_UNUSED (C macro), 575
ESP_BLE_MESH_MODEL_LIGHT_CTL_CLI (C macro), 808
ESP_BLE_MESH_MODEL_LIGHT_CTL_SETUP_SRV (C macro), 810
ESP_BLE_MESH_MODEL_LIGHT_CTL_SRV (C macro), 810

ESP_BLE_MESH_MODEL_LIGHT_CTL_TEMP_SRV (*C macro*), 810
ESP_BLE_MESH_MODEL_LIGHT_HSL_CLI (*C macro*), 809
ESP_BLE_MESH_MODEL_LIGHT_HSL_HUE_SRV (*C macro*), 811
ESP_BLE_MESH_MODEL_LIGHT_HSL_SAT_SRV (*C macro*), 811
ESP_BLE_MESH_MODEL_LIGHT_HSL_SETUP_SRV (*C macro*), 811
ESP_BLE_MESH_MODEL_LIGHT_HSL_SRV (*C macro*), 811
ESP_BLE_MESH_MODEL_LIGHT_LC_CLI (*C macro*), 809
ESP_BLE_MESH_MODEL_LIGHT_LC_SETUP_SRV (*C macro*), 813
ESP_BLE_MESH_MODEL_LIGHT_LC_SRV (*C macro*), 812
ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_CLI (*C macro*), 808
ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SETUP_SRV (*C macro*), 810
ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SRV (*C macro*), 809
ESP_BLE_MESH_MODEL_LIGHT_XYL_CLI (*C macro*), 809
ESP_BLE_MESH_MODEL_LIGHT_XYL_SETUP_SRV (*C macro*), 812
ESP_BLE_MESH_MODEL_LIGHT_XYL_SRV (*C macro*), 812
esp_ble_mesh_model_msg_opcode_init (*C++ function*), 612
ESP_BLE_MESH_MODEL_NONE (*C macro*), 578
ESP_BLE_MESH_MODEL_OP (*C macro*), 578
ESP_BLE_MESH_MODEL_OP_1 (*C macro*), 577
ESP_BLE_MESH_MODEL_OP_2 (*C macro*), 577
ESP_BLE_MESH_MODEL_OP_3 (*C macro*), 577
ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_APP_KEY_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_APP_KEY_LIST (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_APP_KEY_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_ATTENTION_GET (*C macro*), 586
ESP_BLE_MESH_MODEL_OP_ATTENTION_SET (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_ATTENTION_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_BEACON_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_BEACON_SET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_BEACON_STATUS (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_STATUS (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_STATUS (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_END (*C macro*), 578
ESP_BLE_MESH_MODEL_OP_FRIEND_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_FRIEND_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_FRIEND_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_GATT_PROXY_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNAC (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATU (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNAC (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET (*C macro*), 587

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_GET (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_STATUS (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_GET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET_UNACK (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNACK (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_GET (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_GET (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK (*C macro*), 589
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS (*C macro*), 588
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_HEALTH_CURRENT_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR (*C macro*), 586
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK (*C macro*), 586
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET (*C macro*), 586
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET (*C macro*), 586
ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET (*C macro*), 587

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_STATUS (*C macro*), 587
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET_UNACK (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET_UNACK (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET_UNACK (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET_UNACK (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET_UNACK (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET_UNACK (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET_UNACK (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_GET (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_STATUS (*C macro*), 594
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_GET (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET_UNACK (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_STATUS (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_GET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_STATUS (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_GET (*C macro*), 596

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET_UNACK (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS (*C macro*), 596
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET_UNACK (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_GET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_GET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_GET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET_UNACK (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_STATUS (*C macro*), 593
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_GET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_STATUS (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_GET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_GET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_STATUS (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET_UNACK (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_STATUS (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_GET (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_STATUS (*C macro*), 595
ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_GET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_APP_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELE (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVER (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE (*C macro*), 584

ESP_BLE_MESH_MODEL_OP_NET_KEY_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_NET_KEY_LIST (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_NET_KEY_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_GET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_NODE_RESET (*C macro*), 584
ESP_BLE_MESH_MODEL_OP_NODE_RESET_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_RELAY_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_RELAY_SET (*C macro*), 583
ESP_BLE_MESH_MODEL_OP_RELAY_STATUS (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_SCENE_DELETE (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SCENE_RECALL (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_GET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_STORE (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_GET (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS (*C macro*), 592
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS (*C macro*), 590
ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_LIST (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_LIST (*C macro*), 585
esp_ble_mesh_model_op_t (*C++ struct*), 565
esp_ble_mesh_model_op_t::min_len (*C++ member*), 565
esp_ble_mesh_model_op_t::opcode (*C++ member*), 565
esp_ble_mesh_model_op_t::param_cb (*C++ member*), 565
ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_SET

(*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_SET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_GET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS (*C macro*), 591
ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_LIST (*C macro*), 585
ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET (*C macro*), 582
ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_LIST (*C macro*), 585
ESP_BLE_MESH_MODEL_PUB_DEFINE (*C macro*), 578
esp_ble_mesh_model_pub_t (*C++ struct*), 564
esp_ble_mesh_model_pub_t::app_idx (*C++ member*), 564
esp_ble_mesh_model_pub_t::count (*C++ member*), 565
esp_ble_mesh_model_pub_t::cred (*C++ member*), 564
esp_ble_mesh_model_pub_t::dev_role (*C++ member*), 565
esp_ble_mesh_model_pub_t::fast_period (*C++ member*), 564
esp_ble_mesh_model_pub_t::model (*C++ member*), 564
esp_ble_mesh_model_pub_t::msg (*C++ member*), 565
esp_ble_mesh_model_pub_t::period (*C++ member*), 564
esp_ble_mesh_model_pub_t::period_div (*C++ member*), 564
esp_ble_mesh_model_pub_t::period_start (*C++ member*), 565
esp_ble_mesh_model_pub_t::publish_addr (*C++ member*), 564
esp_ble_mesh_model_pub_t::retransmit (*C++ member*), 564
esp_ble_mesh_model_pub_t::send_rel (*C++ member*), 564
esp_ble_mesh_model_pub_t::timer (*C++ member*), 565
esp_ble_mesh_model_pub_t::ttl (*C++ member*), 564
esp_ble_mesh_model_pub_t::update (*C++ member*), 565
esp_ble_mesh_model_publish (*C++ function*), 613
ESP_BLE_MESH_MODEL_SCENE_CLI (*C macro*), 753
ESP_BLE_MESH_MODEL_SCENE_SETUP_SRV (*C macro*), 755
ESP_BLE_MESH_MODEL_SCENE_SRV (*C macro*), 754
ESP_BLE_MESH_MODEL_SCHEDULER_CLI (*C macro*), 753
ESP_BLE_MESH_MODEL_SCHEDULER_SETUP_SRV (*C macro*), 755
ESP_BLE_MESH_MODEL_SCHEDULER_SRV (*C macro*), 755
ESP_BLE_MESH_MODEL_SENSOR_CLI (*C macro*), 725
ESP_BLE_MESH_MODEL_SENSOR_SETUP_SRV (*C macro*), 726
ESP_BLE_MESH_MODEL_SENSOR_SRV (*C macro*), 726
ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MAX (*C macro*), 596
ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MIN (*C macro*), 596
ESP_BLE_MESH_MODEL_STATUS_SUCCESS (*C macro*), 596
esp_ble_mesh_model_status_t (*C++ type*), 598
esp_ble_mesh_model_subscribe_group_addr (*C++ function*), 610
esp_ble_mesh_model_t (*C++ type*), 597
ESP_BLE_MESH_MODEL_TIME_CLI (*C macro*), 753
ESP_BLE_MESH_MODEL_TIME_SETUP_SRV (*C macro*), 754
ESP_BLE_MESH_MODEL_TIME_SRV (*C macro*), 754
esp_ble_mesh_model_unsubscribe_group_addr (*C++ function*), 610
esp_ble_mesh_msg_ctx_t (*C++ struct*), 567
esp_ble_mesh_msg_ctx_t::addr (*C++ member*), 567
esp_ble_mesh_msg_ctx_t::app_idx (*C++ member*), 567
esp_ble_mesh_msg_ctx_t::model (*C++ member*), 567
esp_ble_mesh_msg_ctx_t::net_idx (*C++ member*), 567
esp_ble_mesh_msg_ctx_t::recv_dst (*C++ member*), 567
esp_ble_mesh_msg_ctx_t::recv_op (*C++ member*), 567

esp_ble_mesh_msg_ctx_t::recv_rssi (C++ member), 567
esp_ble_mesh_msg_ctx_t::recv_ttl (C++ member), 567
esp_ble_mesh_msg_ctx_t::send_rel (C++ member), 567
esp_ble_mesh_msg_ctx_t::send_ttl (C++ member), 567
esp_ble_mesh_msg_ctx_t::srv_send (C++ member), 567
ESP_BLE_MESH_NET_PRIMARY (C macro), 575
ESP_BLE_MESH_NO_FAULT (C macro), 667
ESP_BLE_MESH_NO_LOAD_ERROR (C macro), 667
ESP_BLE_MESH_NO_LOAD_WARNING (C macro), 667
esp_ble_mesh_node_add_local_app_key (C++ function), 611
esp_ble_mesh_node_add_local_net_key (C++ function), 610
esp_ble_mesh_node_bind_app_key_to_local_model (C++ function), 611
esp_ble_mesh_node_get_local_app_key (C++ function), 610
esp_ble_mesh_node_get_local_net_key (C++ function), 610
ESP_BLE_MESH_NODE_IDENTITY_NOT_SUPPORTED (C macro), 576
ESP_BLE_MESH_NODE_IDENTITY_RUNNING (C macro), 576
ESP_BLE_MESH_NODE_IDENTITY_STOPPED (C macro), 576
esp_ble_mesh_node_input_number (C++ function), 621
esp_ble_mesh_node_input_string (C++ function), 621
esp_ble_mesh_node_is_provisioned (C++ function), 620
esp_ble_mesh_node_local_reset (C++ function), 614
ESP_BLE_MESH_NODE_NAME_MAX_LEN (C macro), 574
esp_ble_mesh_node_prov_disable (C++ function), 620
esp_ble_mesh_node_prov_enable (C++ function), 620
esp_ble_mesh_node_set_oob_pub_key (C++ function), 621
esp_ble_mesh_node_t (C++ struct), 569
esp_ble_mesh_node_t::addr (C++ member), 569
esp_ble_mesh_node_t::addr_type (C++ member), 569
esp_ble_mesh_node_t::comp_data (C++ member), 570
esp_ble_mesh_node_t::comp_length (C++ member), 570
esp_ble_mesh_node_t::dev_key (C++ member), 570
esp_ble_mesh_node_t::dev_uuid (C++ member), 569
esp_ble_mesh_node_t::element_num (C++ member), 569
esp_ble_mesh_node_t::flags (C++ member), 570
esp_ble_mesh_node_t::iv_index (C++ member), 570
esp_ble_mesh_node_t::name (C++ member), 570
esp_ble_mesh_node_t::net_idx (C++ member), 569
esp_ble_mesh_node_t::oob_info (C++ member), 569
esp_ble_mesh_node_t::unicast_addr (C++ member), 569
ESP_BLE_MESH_OCTET16_LEN (C macro), 574
esp_ble_mesh_octet16_t (C++ type), 596
ESP_BLE_MESH_OCTET8_LEN (C macro), 574
esp_ble_mesh_octet8_t (C++ type), 596
esp_ble_mesh_oob_method_t (C++ enum), 599
esp_ble_mesh_oob_method_t::ESP_BLE_MESH_INPUT_OOB (C++ enumerator), 599
esp_ble_mesh_oob_method_t::ESP_BLE_MESH_NO_OOB (C++ enumerator), 599
esp_ble_mesh_oob_method_t::ESP_BLE_MESH_OUTPUT_OO (C++ enumerator), 599
esp_ble_mesh_oob_method_t::ESP_BLE_MESH_STATIC_OO (C++ enumerator), 599
esp_ble_mesh_opcode_config_client_get_t (C++ type), 597
esp_ble_mesh_opcode_config_client_set_t (C++ type), 597
esp_ble_mesh_opcode_config_status_t (C++ type), 597
esp_ble_mesh_opcode_health_client_get_t (C++ type), 597
esp_ble_mesh_opcode_health_client_set_t (C++ type), 597
esp_ble_mesh_opcode_t (C++ type), 598
esp_ble_mesh_output_action_t (C++ enum), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_BEEP (C++ enumerator), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_BLINK (C++ enumerator), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_DISPLA (C++ enumerator), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_DISPLA (C++ enumerator), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_NO_OUT (C++ enumerator), 599
esp_ble_mesh_output_action_t::ESP_BLE_MESH_VIBRAT (C++ enumerator), 599
ESP_BLE_MESH_OVERFLOW_ERROR (C macro), 668
ESP_BLE_MESH_OVERFLOW_WARNING (C macro), 668

ESP_BLE_MESH_OVERHEAT_ERROR (*C macro*), 667
ESP_BLE_MESH_OVERHEAT_WARNING (*C macro*), 667
ESP_BLE_MESH_OVERLOAD_ERROR (*C macro*), 667
ESP_BLE_MESH_OVERLOAD_WARNING (*C macro*), 667
ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_ERROR (*C macro*), 667
ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_WARNING (*C macro*), 667
ESP_BLE_MESH_PROV (*C macro*), 578
esp_ble_mesh_prov_adv_cb_t (*C++ type*), 625
esp_ble_mesh_prov_bearer_t (*C++ enum*), 599
esp_ble_mesh_prov_bearer_t::ESP_BLE_MESH_PROV_ADV (*C++ enumerator*), 600
esp_ble_mesh_prov_bearer_t::ESP_BLE_MESH_PROV_GATT (*C++ enumerator*), 600
esp_ble_mesh_prov_cb_event_t (*C++ enum*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_CLOSE_LINK_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_FRIEND_ESTABLISH_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_FRIEND_TERMINATE_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_RECV_ADV_PKT_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_RECV_LINK_CLOSE_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_RECV_LINK_OPEN_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_RECV_TRANS_ACK_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_SEND_LINK_CLOSE_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_SEND_LINK_OPEN_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_SEND_TRANS_ACK_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PB_ADV_SEND_TRANS_START_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROV_REGISTER_COMP_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUM_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STR_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT (*C++ enumerator*), 601
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SUBSCRIBE_GROUP_ADDR_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CONNECT_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_IDENTITY_ENABLE_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_PROV_ENABLE_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_FILTER_TYPE_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_NET_BEACON_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_PROXY_FILTER_SRC_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_RELAY_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_UNPROV_DEV_NAME_COMP_EVT (*C++ enumerator*), 604

esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UUID_MATCH_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT (*C++ enumerator*), 602
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT (*C++ enumerator*), 603
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT (*C++ enumerator*), 604
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_PROXY_ (*C++ enumerator*), 606
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_FA (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_event_t::ESP_BLE_MESH_SET_FA (*C++ enumerator*), 605
esp_ble_mesh_prov_cb_param_t (*C++ union*), 532
esp_ble_mesh_prov_cb_param_t::ble_mesh_deinit_mes (*C++ struct*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_deinit_mes (*C++ member*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ struct*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ member*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ struct*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ member*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ member*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enum*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enumerator*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enumerator*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enumerator*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enumerator*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (*C++ enumerator*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_ (*C++ struct*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_ (*C++ member*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_ (*C++ member*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt_ (*C++ struct*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt_ (*C++ member*), 541
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt_ (*C++ member*), 541

esp_ble_mesh_prov_cb_param_t::ble_mesh_input_number_comp_param (C++ struct), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_number_comp_param (C++ member), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_string_comp_param (C++ struct), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_input_string_comp_param (C++ member), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close (C++ struct), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close (C++ member), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_open_param (C++ struct), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_open_param (C++ member), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_output_number_param (C++ struct), 542
esp_ble_mesh_prov_cb_param_t::ble_mesh_output_number_param (C++ member), 543
esp_ble_mesh_prov_cb_param_t::ble_mesh_output_string_param (C++ struct), 543
esp_ble_mesh_prov_cb_param_t::ble_mesh_output_string_param (C++ member), 543
esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_complete_param (C++ struct), 543
esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_complete_param (C++ member), 543
esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_register_comp_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_register_comp_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_unprov_dev_name_comp_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_unprov_dev_name_comp_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_unprov_dev_add_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_unprov_dev_add_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_ct_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_ct_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_pub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_pub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_state_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_local_state_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_ct_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_ct_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_pub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_pub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_state_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_update_remote_state_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_model_unsub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_model_unsub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_publish_comp_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_publish_comp_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_publish_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_publish_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_sub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_unsub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vendor_unsub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_operation_comp_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_operation_comp_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_operation_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_operation_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_publish_comp_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_publish_comp_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_publish_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_publish_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_pub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_pub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ struct), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous] (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_status (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::sub_type (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_id (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_model_instance (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_publish_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_addr (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_update_sub_param::[anonymous]::vnd_sub_opcode (C++ member), 544
esp_ble_mesh_prov_cb_param_t::ble_mesh_vnd_model_update_sub_param::[anonymous

esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 546
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 546
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 546
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 546
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 547
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ struct*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548
esp_ble_mesh_prov_cb_param_t::ble_mesh_[illegible]
(*C++ member*), 548

[illegible]
(*C++ struct*), 548
[illegible]
(*C++ member*), 549
[illegible]
(*C++ struct*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ struct*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ struct*), 549
[illegible]
(*C++ member*), 549
[illegible]
(*C++ struct*), 549
[illegible]
(*C++ member*), 550
[illegible]
(*C++ struct*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ struct*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ struct*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ struct*), 550
[illegible]
(*C++ member*), 550
[illegible]
(*C++ member*), 551
[illegible]
(*C++ member*), 550
[illegible]
(*C++ member*), 551
[illegible]
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ struct*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ struct*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisione
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 551

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 552

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_clie
(*C++ struct*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_output_evt_param::string
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_output_evt_param::[anonymous]
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_comp_param::err_code
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_comp_param
(*C++ struct*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_evt_param
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_evt_param::link_idx
(*C++ member*), 553

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param
(*C++ struct*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::addr
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::addr_type
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::adv_type
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::bearer
(*C++ struct*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::dev_uuid
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::oob_info
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_recv_unprov_adv_pkt_param::rssi
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_dev_uuid_match_comp_param
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_dev_uuid_match_comp_param::err_code
(*C++ struct*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_node_name_comp_param
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_node_name_comp_param::err_code
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_node_name_comp_param::node_index
(*C++ member*), 554

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_primary_elem_addr_comp_param
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_primary_elem_addr_comp_param::err_code
(*C++ struct*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_prov_data_info_comp_param
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_prov_data_info_comp_param::err_code
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_static_oob_val_comp_param
(*C++ struct*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_static_oob_val_comp_param::err_code
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_store_node_comp_data_comp_param
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_store_node_comp_data_comp_param::addr
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_store_node_comp_data_comp_param::err_code
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_[illegible]
(*C++ member*), 555

esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_recv_adv_pkt_param (*C++ struct*), 555
esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_recv_adv_pkt_param::addr (*C++ member*), 555
esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_recv_adv_pkt_param::addr_type (*C++ member*), 555
esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_recv_adv_pkt_param::net_idx (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_recv_adv_pkt_param::rssi (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param (*C++ struct*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_unicast (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_net_idx (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_flags (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_iv_index (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_role (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_primary (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_subnet (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_flag (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_iv_update (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_role (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param::status_primary (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_remove_local_ongoing_adv_comp_param (*C++ struct*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_remove_local_ongoing_adv_comp_param::index (*C++ member*), 540
esp_ble_mesh_prov_cb_param_t::ble_mesh_remove_local_ongoing_adv_comp_param::index (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_remove_local_ongoing_adv_comp_param::err_code (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_remove_local_ongoing_adv_comp_param::net_idx (*C++ member*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param (*C++ struct*), 556
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param::pub_key_x (*C++ member*), 539
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param::pub_key_x (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param::index (*C++ member*), 537
esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param::net_idx (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param (*C++ struct*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::hb_src (*C++ member*), 538
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::hb_dst (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::init_ttl (*C++ member*), 537
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::hops (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::feature (*C++ struct*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::err_code (*C++ member*), 538
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::index (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_recv_param::net_idx (*C++ member*), 537
esp_ble_mesh_prov_cb_param_t::ble_mesh_node_identity_enable_comp_param (*C++ struct*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_node_identity_enable_comp_param::net_idx (*C++ member*), 538
esp_ble_mesh_prov_cb_param_t::ble_mesh_node_identity_enable_comp_param::err_code (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_establish_comp_param (*C++ struct*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_establish_comp_param::net_idx (*C++ member*), 539
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_terminate_comp_param (*C++ member*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_terminate_comp_param::net_idx (*C++ member*), 539
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_poll_comp_param (*C++ struct*), 557
esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_poll_comp_param::net_idx (*C++ member*), 538

esp_ble_mesh_prov_cb_param_t::model_sub (C++ member), 540
esp_ble_mesh_prov_cb_param_t::model_unsub (C++ member), 540
esp_ble_mesh_prov_cb_param_t::node_add (C++ member), 534
esp_ble_mesh_prov_cb_param_t::node_add_net_key (C++ member), 534
esp_ble_mesh_prov_cb_param_t::node_bind (C++ member), 534
esp_ble_mesh_prov_cb_param_t::node_prov_complete (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_disable (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_enable (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_input (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_input_num (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_input_str (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_link_close (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_link_open (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_output (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_output_num (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_output_str (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_reset (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_prov_set_oob_pub_key (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_proxy_gatt_disable (C++ member), 534
esp_ble_mesh_prov_cb_param_t::node_proxy_gatt_enable (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_proxy_identity (C++ member), 533
esp_ble_mesh_prov_cb_param_t::node_set_name (C++ member), 532
esp_ble_mesh_prov_cb_param_t::op (C++ member), 537
esp_ble_mesh_prov_cb_param_t::prov_register_comp (C++ member), 532
esp_ble_mesh_prov_cb_param_t::provisioner_add_app_key (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_add_mesh_key (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_add_unprov_dev (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_bind_app_key_to_model (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_close_multiple_pbg (C++ member), 538
esp_ble_mesh_prov_cb_param_t::provisioner_close_multiple_pbg_with_reason (C++ member), 538
esp_ble_mesh_prov_cb_param_t::provisioner_delete_app_key_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_delete_dev_comp (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_delete_mesh_key_comp (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_delete_node_with_uuid_comp (C++ member), 538
esp_ble_mesh_prov_cb_param_t::provisioner_direct_close_link_comp (C++ member), 537
esp_ble_mesh_prov_cb_param_t::provisioner_enable_comp (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_open_session_comp (C++ member), 537
esp_ble_mesh_prov_cb_param_t::provisioner_open_session_with_uuid_comp (C++ member), 538
esp_ble_mesh_prov_cb_param_t::provisioner_prov_complete (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_device_resume_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_prov_disable_comp (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_enable_comp (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_input (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_input_num (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_prov_input_str (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_prov_link_close (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_link_open (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_output (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_output_num (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_output_str (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_read_oob_pub_key (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_prov_set_node_name_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_recv_heartbeat (C++ member), 537
esp_ble_mesh_prov_cb_param_t::provisioner_recv_unprov_adv_pkt (C++ member), 534
esp_ble_mesh_prov_cb_param_t::provisioner_set_dev_uuid_match_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_set_heartbeat_filter_comp (C++ member), 537
esp_ble_mesh_prov_cb_param_t::provisioner_set_heartbeat_filter_type_comp (C++ member), 537
esp_ble_mesh_prov_cb_param_t::provisioner_set_node_name_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_set_primary_elem_addr_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_set_prov_data_info_comp (C++ member), 535

esp_ble_mesh_prov_cb_param_t::provisioner_set_node_name_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_store_node_comp_data_comp (C++ member), 536
esp_ble_mesh_prov_cb_param_t::provisioner_update_comp (C++ member), 535
esp_ble_mesh_prov_cb_param_t::provisioner_update_node_info_comp (C++ member), 536
esp_ble_mesh_prov_cb_param_t::proxy_client_add_filter_comp (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_connect_comp (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_connected (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_disconnect_comp (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_disconnected (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_recv_adv_pkt (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_recv_filter_status (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_remove_filter_comp (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_client_set_filter_type_comp (C++ member), 539
esp_ble_mesh_prov_cb_param_t::proxy_server_connected (C++ member), 540
esp_ble_mesh_prov_cb_param_t::proxy_server_disconnected (C++ member), 540
esp_ble_mesh_prov_cb_param_t::rssi (C++ member), 537
esp_ble_mesh_prov_cb_param_t::rx_ttl (C++ member), 537
esp_ble_mesh_prov_cb_param_t::set_fast_prov_action_comp (C++ member), 538
esp_ble_mesh_prov_cb_param_t::set_fast_prov_info_comp (C++ member), 538
esp_ble_mesh_prov_cb_param_t::type (C++ member), 536
esp_ble_mesh_prov_cb_param_t::uid (C++ member), 538
esp_ble_mesh_prov_cb_t (C++ type), 625
esp_ble_mesh_prov_data_info_t (C++ struct), 569
esp_ble_mesh_prov_data_info_t::flag (C++ member), 569
esp_ble_mesh_prov_data_info_t::flags (C++ member), 569
esp_ble_mesh_prov_data_info_t::iv_index (C++ member), 569
esp_ble_mesh_prov_data_info_t::net_idx (C++ member), 569
ESP_BLE_MESH_PROV_INPUT_OOB_MAX_LEN (C macro), 577
esp_ble_mesh_prov_oob_info_t (C++ enum), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_ON_BOX (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_ON_DEV (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_OTHER (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_PUB_KEY (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_STATIC (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_STR (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_VAL_ACCEPT (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_VAL_DIGIT (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_VAL_NUM (C++ enumerator), 600
esp_ble_mesh_prov_oob_info_t::ESP_BLE_MESH_PROV_OOB_VAL_STR (C++ enumerator), 600
ESP_BLE_MESH_PROV_OUTPUT_OOB_MAX_LEN (C macro), 577
ESP_BLE_MESH_PROV_STATIC_OOB_MAX_LEN (C macro), 577
esp_ble_mesh_prov_t (C++ struct), 567
esp_ble_mesh_provisioner_add_local_app_key (C++ function), 616
esp_ble_mesh_provisioner_add_local_net_key (C++ function), 616
esp_ble_mesh_provisioner_add_unprov_dev (C++ function), 623
esp_ble_mesh_provisioner_bind_app_key_to_local_model (C++ function), 616
esp_ble_mesh_provisioner_close_settings_with_index (C++ function), 618
esp_ble_mesh_provisioner_close_settings_with_uid (C++ function), 619
esp_ble_mesh_provisioner_delete_dev (C++ function), 624
esp_ble_mesh_provisioner_delete_node_with_addr (C++ function), 615
esp_ble_mesh_provisioner_delete_node_with_uuid (C++ function), 615
esp_ble_mesh_provisioner_delete_settings_with_index (C++ function), 619
esp_ble_mesh_provisioner_delete_settings_with_uid (C++ function), 619
esp_ble_mesh_provisioner_direct_erase_settings (C++ function), 618
esp_ble_mesh_provisioner_get_free_settings_count (C++ function), 620
esp_ble_mesh_provisioner_get_local_app_key (C++ function), 616
esp_ble_mesh_provisioner_get_local_net_key (C++ function), 617

esp_ble_mesh_provisioner_get_node_index (*C++ function*), 614
esp_ble_mesh_provisioner_get_node_name (*C++ function*), 614
esp_ble_mesh_provisioner_get_node_table_entry (*C++ function*), 615
esp_ble_mesh_provisioner_get_node_with_addr (*C++ function*), 615
esp_ble_mesh_provisioner_get_node_with_name (*C++ function*), 615
esp_ble_mesh_provisioner_get_node_with_uuid (*C++ function*), 615
esp_ble_mesh_provisioner_get_prov_node_count (*C++ function*), 615
esp_ble_mesh_provisioner_get_settings_index (*C++ function*), 620
esp_ble_mesh_provisioner_get_settings_uuid (*C++ function*), 619
esp_ble_mesh_provisioner_input_number (*C++ function*), 622
esp_ble_mesh_provisioner_input_string (*C++ function*), 622
esp_ble_mesh_provisioner_open_settings_with_index (*C++ function*), 618
esp_ble_mesh_provisioner_open_settings_with_uuid (*C++ function*), 618
esp_ble_mesh_provisioner_prov_device_with_addr (*C++ function*), 623
esp_ble_mesh_provisioner_prov_disable (*C++ function*), 622
esp_ble_mesh_provisioner_prov_enable (*C++ function*), 622
esp_ble_mesh_provisioner_read_oob_pub_key (*C++ function*), 621
esp_ble_mesh_provisioner_recv_heartbeat (*C++ function*), 617
esp_ble_mesh_provisioner_set_dev_uuid_match (*C++ function*), 624
esp_ble_mesh_provisioner_set_heartbeat_filter_info (*C++ function*), 617
esp_ble_mesh_provisioner_set_heartbeat_filter_type (*C++ function*), 617
esp_ble_mesh_provisioner_set_node_name (*C++ function*), 614
esp_ble_mesh_provisioner_set_primary_elem_addr (*C++ function*), 625
esp_ble_mesh_provisioner_set_prov_data_info (*C++ function*), 624
esp_ble_mesh_provisioner_set_static_oob_value (*C++ function*), 625
esp_ble_mesh_provisioner_store_node_comp_data (*C++ function*), 615
esp_ble_mesh_provisioner_update_local_app_key (*C++ function*), 616
esp_ble_mesh_provisioner_update_local_net_key (*C++ function*), 617
esp_ble_mesh_proxy_client_add_filter_addr (*C++ function*), 627
esp_ble_mesh_proxy_client_connect (*C++ function*), 626
esp_ble_mesh_proxy_client_disconnect (*C++ function*), 626
esp_ble_mesh_proxy_client_remove_filter_addr (*C++ function*), 627
esp_ble_mesh_proxy_client_set_filter_type (*C++ function*), 627
esp_ble_mesh_proxy_filter_type_t (*C++ enum*), 601
esp_ble_mesh_proxy_filter_type_t::PROXY_FILTER_BL (*C++ enumerator*), 601
esp_ble_mesh_proxy_filter_type_t::PROXY_FILTER_WH (*C++ enumerator*), 601
esp_ble_mesh_proxy_gatt_disable (*C++ function*), 626
esp_ble_mesh_proxy_gatt_enable (*C++ function*), 626
esp_ble_mesh_proxy_identity_enable (*C++ function*), 626
ESP_BLE_MESH_PUBLISH_TRANSMIT (*C macro*), 577
esp_ble_mesh_register_config_client_callback (*C++ function*), 628
esp_ble_mesh_register_config_server_callback (*C++ function*), 628
esp_ble_mesh_register_custom_model_callback (*C++ function*), 612
esp_ble_mesh_register_generic_client_callback (*C++ function*), 670
esp_ble_mesh_register_generic_server_callback (*C++ function*), 671
esp_ble_mesh_register_health_client_callback (*C++ function*), 659
esp_ble_mesh_register_health_server_callback (*C++ function*), 659
esp_ble_mesh_register_light_client_callback (*C++ function*), 759
esp_ble_mesh_register_lighting_server_callback (*C++ function*), 759
esp_ble_mesh_register_prov_callback (*C++ function*), 620
esp_ble_mesh_register_sensor_client_callback (*C++ function*), 710
esp_ble_mesh_register_sensor_server_callback (*C++ function*), 711
esp_ble_mesh_register_time_scene_client_callback (*C++ function*), 730
esp_ble_mesh_register_time_scene_server_callback (*C++ function*), 731
ESP_BLE_MESH_RELAY_DISABLED (*C macro*), 575
ESP_BLE_MESH_RELAY_ENABLED (*C macro*), 575
ESP_BLE_MESH_RELAY_NOT_SUPPORTED (*C macro*), 575
esp_ble_mesh_scene_delete_t (*C++ struct*), 736
esp_ble_mesh_scene_delete_t::scene_number

(C++ member), 737
ESP_BLE_MESH_SCENE_NOT_FOUND (C macro), 757
ESP_BLE_MESH_SCENE_NUMBER_LEN (C macro), 756
esp_ble_mesh_scene_recall_t (C++ struct), 736
esp_ble_mesh_scene_recall_t::delay (C++ member), 736
esp_ble_mesh_scene_recall_t::op_en (C++ member), 736
esp_ble_mesh_scene_recall_t::scene_number (C++ member), 736
esp_ble_mesh_scene_recall_t::tid (C++ member), 736
esp_ble_mesh_scene_recall_t::trans_time (C++ member), 736
ESP_BLE_MESH_SCENE_REG_FULL (C macro), 757
esp_ble_mesh_scene_register_status_cb_t (C++ struct), 739
esp_ble_mesh_scene_register_status_cb_t::current_scene (C++ member), 740
esp_ble_mesh_scene_register_status_cb_t::scenes (C++ member), 740
esp_ble_mesh_scene_register_status_cb_t::status_code (C++ member), 740
esp_ble_mesh_scene_register_t (C++ struct), 742
esp_ble_mesh_scene_register_t::scene_number (C++ member), 743
esp_ble_mesh_scene_register_t::scene_type (C++ member), 743
esp_ble_mesh_scene_register_t::scene_value (C++ member), 743
esp_ble_mesh_scene_setup_srv_t (C++ struct), 744
esp_ble_mesh_scene_setup_srv_t::model (C++ member), 745
esp_ble_mesh_scene_setup_srv_t::rsp_ctrl (C++ member), 745
esp_ble_mesh_scene_setup_srv_t::state (C++ member), 745
esp_ble_mesh_scene_srv_t (C++ struct), 744
esp_ble_mesh_scene_srv_t::last (C++ member), 744
esp_ble_mesh_scene_srv_t::model (C++ member), 744
esp_ble_mesh_scene_srv_t::rsp_ctrl (C++ member), 744
esp_ble_mesh_scene_srv_t::state (C++ member), 744
esp_ble_mesh_scene_srv_t::transition (C++ member), 744
esp_ble_mesh_scene_status_cb_t (C++ struct), 739
esp_ble_mesh_scene_status_cb_t::current_scene (C++ member), 739
esp_ble_mesh_scene_status_cb_t::op_en (C++ member), 739
esp_ble_mesh_scene_status_cb_t::remain_time (C++ member), 739
esp_ble_mesh_scene_status_cb_t::status_code (C++ member), 739
esp_ble_mesh_scene_status_cb_t::target_scene (C++ member), 739
esp_ble_mesh_scene_store_t (C++ struct), 736
esp_ble_mesh_scene_store_t::scene_number (C++ member), 736
ESP_BLE_MESH_SCENE_SUCCESS (C macro), 757
esp_ble_mesh_scenes_state_t (C++ struct), 743
esp_ble_mesh_scenes_state_t::current_scene (C++ member), 743
esp_ble_mesh_scenes_state_t::in_progress (C++ member), 744
esp_ble_mesh_scenes_state_t::scene_count (C++ member), 743
esp_ble_mesh_scenes_state_t::scenes (C++ member), 743
esp_ble_mesh_scenes_state_t::status_code (C++ member), 744
esp_ble_mesh_scenes_state_t::target_scene (C++ member), 744
ESP_BLE_MESH_SCHEDULE_ACT_NO_ACTION (C macro), 757
ESP_BLE_MESH_SCHEDULE_ACT_SCENE_RECALL (C macro), 757
ESP_BLE_MESH_SCHEDULE_ACT_TURN_OFF (C macro), 757
ESP_BLE_MESH_SCHEDULE_ACT_TURN_ON (C macro), 757
ESP_BLE_MESH_SCHEDULE_DAY_ANY_DAY (C macro), 756
ESP_BLE_MESH_SCHEDULE_ENTRY_MAX_INDEX (C macro), 757
ESP_BLE_MESH_SCHEDULE_HOUR_ANY_HOUR (C macro), 756
ESP_BLE_MESH_SCHEDULE_HOUR_ONCE_A_DAY (C macro), 756
esp_ble_mesh_schedule_register_t (C++ struct), 745
esp_ble_mesh_schedule_register_t::action (C++ member), 745
esp_ble_mesh_schedule_register_t::day (C++ member), 745
esp_ble_mesh_schedule_register_t::day_of_week (C++ member), 745
esp_ble_mesh_schedule_register_t::hour (C++ member), 745
esp_ble_mesh_schedule_register_t::in_use (C++ member), 745
esp_ble_mesh_schedule_register_t::minute (C++ member), 745
esp_ble_mesh_schedule_register_t::month

(C++ *member*), 745
esp_ble_mesh_schedule_register_t::scene_number (C++ *member*), 745
esp_ble_mesh_schedule_register_t::second (C++ *member*), 745
esp_ble_mesh_schedule_register_t::trans_time (C++ *member*), 745
esp_ble_mesh_schedule_register_t::year (C++ *member*), 745
ESP_BLE_MESH_SCHEDULE_SCENE_NO_SCENE (*C macro*), 757
ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_HOUR (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_MIN (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_EVERY_15_MIN (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_EVERY_15_SEC (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_EVERY_20_MIN (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_EVERY_20_SEC (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_HOUR (*C macro*), 756
ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_MIN (*C macro*), 757
ESP_BLE_MESH_SCHEDULE_YEAR_ANY_YEAR (*C macro*), 756
esp_ble_mesh_scheduler_act_get_t (*C++ struct*), 737
esp_ble_mesh_scheduler_act_get_t::index (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t (*C++ struct*), 737
esp_ble_mesh_scheduler_act_set_t::action (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::day (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::day_of_week (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::hour (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::index (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::minute (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::month (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::scene_number (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::second (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::trans_time (C++ *member*), 737
esp_ble_mesh_scheduler_act_set_t::year (C++ *member*), 737
esp_ble_mesh_scheduler_act_status_cb_t (*C++ struct*), 740
esp_ble_mesh_scheduler_act_status_cb_t::action (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::day (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::day_of_week (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::hour (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::index (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::minute (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::month (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::scene_number (C++ *member*), 741
esp_ble_mesh_scheduler_act_status_cb_t::second (C++ *member*), 740
esp_ble_mesh_scheduler_act_status_cb_t::trans_time (C++ *member*), 741
esp_ble_mesh_scheduler_act_status_cb_t::year (C++ *member*), 740
esp_ble_mesh_scheduler_setup_srv_t (*C++ struct*), 746
esp_ble_mesh_scheduler_setup_srv_t::model (C++ *member*), 746
esp_ble_mesh_scheduler_setup_srv_t::rsp_ctrl (C++ *member*), 746
esp_ble_mesh_scheduler_setup_srv_t::state (C++ *member*), 746
esp_ble_mesh_scheduler_srv_t (*C++ struct*), 746
esp_ble_mesh_scheduler_srv_t::model (C++ *member*), 746
esp_ble_mesh_scheduler_srv_t::rsp_ctrl (C++ *member*), 746
esp_ble_mesh_scheduler_srv_t::state (C++ *member*), 746
esp_ble_mesh_scheduler_state_t (*C++ struct*), 745
esp_ble_mesh_scheduler_state_t::schedule_count (C++ *member*), 746
esp_ble_mesh_scheduler_state_t::schedules (C++ *member*), 746
esp_ble_mesh_scheduler_status_cb_t (*C++ struct*), 740
esp_ble_mesh_scheduler_status_cb_t::schedules (C++ *member*), 740
ESP_BLE_MESH_SDU_MAX_LEN (*C macro*), 574
ESP_BLE_MESH_SELF_TEST_ERROR (*C macro*), 668
ESP_BLE_MESH_SELF_TEST_WARNING (*C macro*), 668
esp_ble_mesh_sensor_cadence_get_t (*C++ struct*), 714
esp_ble_mesh_sensor_cadence_get_t::property_id (C++ *member*), 714

esp_ble_mesh_sensor_cadence_set_t (C++ *struct*), 714
esp_ble_mesh_sensor_cadence_set_t::fast_cadence_high (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::fast_cadence_low (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::fast_cadence_period_divisor (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::property_id (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::status_bit (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::trigger_delta_down (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::trigger_delta_up (C++ *member*), 715
esp_ble_mesh_sensor_cadence_set_t::trigger_type (C++ *member*), 715
esp_ble_mesh_sensor_cadence_status_cb_t (C++ *struct*), 717
esp_ble_mesh_sensor_cadence_status_cb_t::property_id (C++ *member*), 717
esp_ble_mesh_sensor_cadence_status_cb_t::sensor_cadence_value (C++ *member*), 717
esp_ble_mesh_sensor_cadence_t (C++ *struct*), 719
esp_ble_mesh_sensor_cadence_t::fast_cadence_high (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::fast_cadence_low (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::min_interval (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::period_divisor (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::trigger_delta_down (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::trigger_delta_up (C++ *member*), 720
esp_ble_mesh_sensor_cadence_t::trigger_type (C++ *member*), 720
esp_ble_mesh_sensor_client_cb_event_t (C++ *enum*), 729
esp_ble_mesh_sensor_client_cb_event_t::ESP_BLE_MESH_SENSOR_CLIENT_EVT_MAX (C++ *enumerator*), 729
esp_ble_mesh_sensor_client_cb_event_t::ESP_BLE_MESH_SENSOR_CLIENT_GET_STATE_EVT (C++ *enumerator*), 729
esp_ble_mesh_sensor_client_cb_event_t::ESP_BLE_MESH_SENSOR_CLIENT_PUBLISH_EVT (C++ *enumerator*), 729
esp_ble_mesh_sensor_client_cb_event_t::ESP_BLE_MESH_SENSOR_CLIENT_SET_STATE_EVT (C++ *enumerator*), 729
esp_ble_mesh_sensor_client_cb_event_t::ESP_BLE_MESH_SENSOR_CLIENT_TIMEOUT_EVT (C++ *enumerator*), 729
esp_ble_mesh_sensor_client_cb_param_t (C++ *struct*), 718
esp_ble_mesh_sensor_client_cb_param_t::error_code (C++ *member*), 718
esp_ble_mesh_sensor_client_cb_param_t::params (C++ *member*), 719
esp_ble_mesh_sensor_client_cb_param_t::status_cb (C++ *member*), 719
esp_ble_mesh_sensor_client_cb_t (C++ *type*), 729
esp_ble_mesh_sensor_client_get_state (C++ *function*), 710
esp_ble_mesh_sensor_client_get_state_t (C++ *union*), 711
esp_ble_mesh_sensor_client_get_state_t::cadence_get (C++ *member*), 711
esp_ble_mesh_sensor_client_get_state_t::column_get (C++ *member*), 712
esp_ble_mesh_sensor_client_get_state_t::descriptor_get (C++ *member*), 711
esp_ble_mesh_sensor_client_get_state_t::sensor_get (C++ *member*), 712
esp_ble_mesh_sensor_client_get_state_t::series_get (C++ *member*), 712
esp_ble_mesh_sensor_client_get_state_t::setting_get (C++ *member*), 711
esp_ble_mesh_sensor_client_get_state_t::settings_get (C++ *member*), 711
esp_ble_mesh_sensor_client_set_state (C++ *function*), 711
esp_ble_mesh_sensor_client_set_state_t (C++ *union*), 712
esp_ble_mesh_sensor_client_set_state_t::cadence_set (C++ *member*), 712
esp_ble_mesh_sensor_client_set_state_t::setting_set (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t (C++ *union*), 712
esp_ble_mesh_sensor_client_status_cb_t::cadence_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::column_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::descriptor_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::sensor_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::series_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::setting_status (C++ *member*), 712
esp_ble_mesh_sensor_client_status_cb_t::settings_status (C++ *member*), 712
esp_ble_mesh_sensor_column_get_cb_t (C++ *struct*), 716
esp_ble_mesh_sensor_column_get_cb_t::property_id (C++ *member*), 716
esp_ble_mesh_sensor_column_get_cb_t::raw_value_x (C++ *member*), 716
esp_ble_mesh_sensor_column_status_cb_t (C++ *struct*), 718
esp_ble_mesh_sensor_column_status_cb_t::property_id (C++ *member*), 718
esp_ble_mesh_sensor_column_status_cb_t::sensor_column_value (C++ *member*), 718

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A (*C macro*), 727
ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID (*C macro*), 728
ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID_LEN (*C macro*), 727
ESP_BLE_MESH_SENSOR_DATA_FORMAT_B (*C macro*), 727
ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID (*C macro*), 728
ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN (*C macro*), 727
esp_ble_mesh_sensor_data_t (*C++ struct*), 720
esp_ble_mesh_sensor_data_t::format (*C++ member*), 720
esp_ble_mesh_sensor_data_t::length (*C++ member*), 720
esp_ble_mesh_sensor_data_t::raw_value (*C++ member*), 720
ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN (*C macro*), 727
esp_ble_mesh_sensor_descriptor_get_t (*C++ struct*), 714
esp_ble_mesh_sensor_descriptor_get_t::op_en (*C++ member*), 714
esp_ble_mesh_sensor_descriptor_get_t::property_id (*C++ member*), 714
ESP_BLE_MESH_SENSOR_DESCRIPTOR_LEN (*C macro*), 726
esp_ble_mesh_sensor_descriptor_status_cb_t (*C++ struct*), 717
esp_ble_mesh_sensor_descriptor_status_cb_t::descriptor (*C++ member*), 717
esp_ble_mesh_sensor_descriptor_t (*C++ struct*), 719
esp_ble_mesh_sensor_descriptor_t::measure_period (*C++ member*), 719
esp_ble_mesh_sensor_descriptor_t::negative_tolerance (*C++ member*), 719
esp_ble_mesh_sensor_descriptor_t::positive_tolerance (*C++ member*), 719
esp_ble_mesh_sensor_descriptor_t::sampling_function (*C++ member*), 719
esp_ble_mesh_sensor_descriptor_t::update_interval (*C++ member*), 719
ESP_BLE_MESH_SENSOR_DIVISOR_TRIGGER_TYPE_LEN (*C macro*), 727
esp_ble_mesh_sensor_get_t (*C++ struct*), 716
esp_ble_mesh_sensor_get_t::op_en (*C++ member*), 716
esp_ble_mesh_sensor_get_t::property_id (*C++ member*), 716
esp_ble_mesh_sensor_message_opcode_t (*C++ type*), 598
ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD (*C macro*), 726
ESP_BLE_MESH_SENSOR_NOT_APPL_UPDATE_INTERVAL (*C macro*), 726
ESP_BLE_MESH_SENSOR_PERIOD_DIVISOR_MAX_VALUE (*C macro*), 727
ESP_BLE_MESH_SENSOR_PROPERTY_ID_LEN (*C macro*), 726
esp_ble_mesh_sensor_sample_func (*C++ enum*), 729
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 729
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 730
esp_ble_mesh_sensor_sample_func::ESP_BLE_MESH_SAM (*C++ enumerator*), 729
esp_ble_mesh_sensor_series_column_t (*C++ struct*), 720
esp_ble_mesh_sensor_series_column_t::column_width (*C++ member*), 721
esp_ble_mesh_sensor_series_column_t::raw_value_x (*C++ member*), 720
esp_ble_mesh_sensor_series_column_t::raw_value_y (*C++ member*), 721
esp_ble_mesh_sensor_series_get_t (*C++ struct*), 716
esp_ble_mesh_sensor_series_get_t::op_en (*C++ member*), 716
esp_ble_mesh_sensor_series_get_t::property_id (*C++ member*), 716
esp_ble_mesh_sensor_series_get_t::raw_value_x1 (*C++ member*), 716
esp_ble_mesh_sensor_series_get_t::raw_value_x2 (*C++ member*), 716
esp_ble_mesh_sensor_series_status_cb_t (*C++ struct*), 718
esp_ble_mesh_sensor_series_status_cb_t::property_ (*C++ member*), 718
esp_ble_mesh_sensor_series_status_cb_t::sensor_se (*C++ member*), 718
esp_ble_mesh_sensor_server_cb_event_t (*C++ enum*), 730
esp_ble_mesh_sensor_server_cb_event_t::ESP_BLE_ME (*C++ enumerator*), 730
esp_ble_mesh_sensor_server_cb_event_t::ESP_BLE_ME (*C++ enumerator*), 730
esp_ble_mesh_sensor_server_cb_event_t::ESP_BLE_ME (*C++ enumerator*), 730
esp_ble_mesh_sensor_server_cb_event_t::ESP_BLE_ME (*C++ enumerator*), 730
esp_ble_mesh_sensor_server_cb_param_t

(C++ *struct*), 725
esp_ble_mesh_sensor_server_cb_param_t::ctx (C++ *member*), 725
esp_ble_mesh_sensor_server_cb_param_t::model (C++ *member*), 725
esp_ble_mesh_sensor_server_cb_param_t::value (C++ *member*), 725
esp_ble_mesh_sensor_server_cb_t (C++ *type*), 729
esp_ble_mesh_sensor_server_cb_value_t (C++ *union*), 714
esp_ble_mesh_sensor_server_cb_value_t::get (C++ *member*), 714
esp_ble_mesh_sensor_server_cb_value_t::set (C++ *member*), 714
esp_ble_mesh_sensor_server_cb_value_t::state_change (C++ *member*), 714
esp_ble_mesh_sensor_server_recv_get_msg_t (C++ *union*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_cadence_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_column_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_descriptor_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_series_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_setting_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_get_msg_t::bles_mesh_settings_get (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_set_msg_t (C++ *union*), 713
esp_ble_mesh_sensor_server_recv_set_msg_t::bles_mesh_cadence_set (C++ *member*), 713
esp_ble_mesh_sensor_server_recv_set_msg_t::bles_mesh_setting_set (C++ *member*), 714
esp_ble_mesh_sensor_server_state_change_t (C++ *union*), 713
esp_ble_mesh_sensor_server_state_change_t::bles_mesh_cadence_status (C++ *member*), 713
esp_ble_mesh_sensor_server_state_change_t::bles_mesh_setting_status (C++ *member*), 713
ESP_BLE_MESH_SENSOR_SETTING_ACCESS_LEN (C *macro*), 727
ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ (C *macro*), 727
ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ_WRITE (C *macro*), 727
esp_ble_mesh_sensor_setting_get_t (C++ *struct*), 715
esp_ble_mesh_sensor_setting_get_t::sensor_property_id (C++ *member*), 715
esp_ble_mesh_sensor_setting_get_t::sensor_setting_property_id (C++ *member*), 715
ESP_BLE_MESH_SENSOR_SETTING_PROPERTY_ID_LEN (C *macro*), 727
esp_ble_mesh_sensor_setting_set_t (C++ *struct*), 715
esp_ble_mesh_sensor_setting_set_t::sensor_property_id (C++ *member*), 716
esp_ble_mesh_sensor_setting_set_t::sensor_setting_len (C++ *member*), 716
esp_ble_mesh_sensor_setting_set_t::sensor_setting_raw (C++ *member*), 716
esp_ble_mesh_sensor_setting_status_cb_t (C++ *struct*), 717
esp_ble_mesh_sensor_setting_status_cb_t::op_en (C++ *member*), 717
esp_ble_mesh_sensor_setting_status_cb_t::sensor_property_id (C++ *member*), 717
esp_ble_mesh_sensor_setting_status_cb_t::sensor_setting_len (C++ *member*), 718
esp_ble_mesh_sensor_setting_status_cb_t::sensor_setting_raw (C++ *member*), 717
esp_ble_mesh_sensor_setting_status_cb_t::sensor_setting_status (C++ *member*), 718
esp_ble_mesh_sensor_setting_t (C++ *struct*), 719
esp_ble_mesh_sensor_setting_t::access (C++ *member*), 719
esp_ble_mesh_sensor_setting_t::property_id (C++ *member*), 719
esp_ble_mesh_sensor_setting_t::raw (C++ *member*), 719
esp_ble_mesh_sensor_settings_get_t (C++ *struct*), 715
esp_ble_mesh_sensor_settings_get_t::sensor_property_id (C++ *member*), 715
esp_ble_mesh_sensor_settings_status_cb_t (C++ *struct*), 717
esp_ble_mesh_sensor_settings_status_cb_t::sensor_property_ids (C++ *member*), 717
esp_ble_mesh_sensor_settings_status_cb_t::sensor_setting_ids (C++ *member*), 717
esp_ble_mesh_sensor_setup_srv_t (C++ *struct*), 722
esp_ble_mesh_sensor_setup_srv_t::model (C++ *member*), 722
esp_ble_mesh_sensor_setup_srv_t::rsp_ctrl (C++ *member*), 722
esp_ble_mesh_sensor_setup_srv_t::state_count (C++ *member*), 722
esp_ble_mesh_sensor_setup_srv_t::states (C++ *member*), 722
esp_ble_mesh_sensor_srv_t (C++ *struct*), 721
esp_ble_mesh_sensor_srv_t::model (C++ *member*), 721
esp_ble_mesh_sensor_srv_t::rsp_ctrl (C++ *member*), 721
esp_ble_mesh_sensor_srv_t::state_count (C++ *member*), 721
esp_ble_mesh_sensor_srv_t::states (C++ *member*), 722

esp_ble_mesh_sensor_state_t (*C++ struct*), 721
esp_ble_mesh_sensor_state_t::cadence (*C++ member*), 721
esp_ble_mesh_sensor_state_t::descriptor (*C++ member*), 721
esp_ble_mesh_sensor_state_t::sensor_data (*C++ member*), 721
esp_ble_mesh_sensor_state_t::sensor_property_id (*C++ member*), 721
esp_ble_mesh_sensor_state_t::series_column (*C++ member*), 721
esp_ble_mesh_sensor_state_t::setting_count (*C++ member*), 721
esp_ble_mesh_sensor_state_t::settings (*C++ member*), 721
esp_ble_mesh_sensor_status_cb_t (*C++ struct*), 718
esp_ble_mesh_sensor_status_cb_t::marshalled_sensor_data (*C++ member*), 718
ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_LEN (*C macro*), 727
ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_MAX (*C macro*), 727
ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_CHAR (*C macro*), 727
ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_UINT16 (*C macro*), 727
ESP_BLE_MESH_SENSOR_UNSPECIFIED_NEG_TOLERANCE (*C macro*), 726
ESP_BLE_MESH_SENSOR_UNSPECIFIED_POS_TOLERANCE (*C macro*), 726
ESP_BLE_MESH_SERVER_AUTO_RSP (*C macro*), 596
esp_ble_mesh_server_model_send_msg (*C++ function*), 613
esp_ble_mesh_server_model_update_state (*C++ function*), 614
esp_ble_mesh_server_recv_gen_admin_property_get (*C++ struct*), 698
esp_ble_mesh_server_recv_gen_admin_property_get::property_id (*C++ member*), 698
esp_ble_mesh_server_recv_gen_admin_property_set (*C++ struct*), 702
esp_ble_mesh_server_recv_gen_admin_property_set::property_id (*C++ member*), 703
esp_ble_mesh_server_recv_gen_admin_property_set::property_value (*C++ member*), 703
esp_ble_mesh_server_recv_gen_admin_property_set::user_access (*C++ member*), 703
esp_ble_mesh_server_recv_gen_client_properties_get (*C++ struct*), 698
esp_ble_mesh_server_recv_gen_client_properties_get::property_id (*C++ member*), 698
esp_ble_mesh_server_recv_gen_def_trans_time_set (*C++ struct*), 700
esp_ble_mesh_server_recv_gen_def_trans_time_set::trans_time (*C++ member*), 700
esp_ble_mesh_server_recv_gen_delta_set_t (*C++ struct*), 699
esp_ble_mesh_server_recv_gen_delta_set_t::delay (*C++ member*), 700
esp_ble_mesh_server_recv_gen_delta_set_t::delta_low (*C++ member*), 700
esp_ble_mesh_server_recv_gen_delta_set_t::op_en (*C++ member*), 699
esp_ble_mesh_server_recv_gen_delta_set_t::tid (*C++ member*), 700
esp_ble_mesh_server_recv_gen_delta_set_t::trans_time (*C++ member*), 700
esp_ble_mesh_server_recv_gen_level_set_t (*C++ struct*), 699
esp_ble_mesh_server_recv_gen_level_set_t::delay (*C++ member*), 699
esp_ble_mesh_server_recv_gen_level_set_t::level (*C++ member*), 699
esp_ble_mesh_server_recv_gen_level_set_t::op_en (*C++ member*), 699
esp_ble_mesh_server_recv_gen_level_set_t::tid (*C++ member*), 699
esp_ble_mesh_server_recv_gen_level_set_t::trans_time (*C++ member*), 699
esp_ble_mesh_server_recv_gen_loc_global_set_t (*C++ struct*), 701
esp_ble_mesh_server_recv_gen_loc_global_set_t::global_north (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_global_set_t::global_altitude (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_global_set_t::global_east (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t (*C++ struct*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t::floor_number (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t::local_altitude (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t::local_east (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t::local_north (*C++ member*), 702
esp_ble_mesh_server_recv_gen_loc_local_set_t::uncertainty (*C++ member*), 702
esp_ble_mesh_server_recv_gen_manufacturer_property_get (*C++ struct*), 698
esp_ble_mesh_server_recv_gen_manufacturer_property_get::property_id (*C++ member*), 698
esp_ble_mesh_server_recv_gen_manufacturer_property_set (*C++ struct*), 703
esp_ble_mesh_server_recv_gen_manufacturer_property_set::property_id (*C++ member*), 703
esp_ble_mesh_server_recv_gen_move_set_t (*C++ struct*), 700
esp_ble_mesh_server_recv_gen_move_set_t::delay (*C++ member*), 700

esp_ble_mesh_server_recv_gen_move_set_t::delta_level (C++ member), 700
esp_ble_mesh_server_recv_gen_move_set_t::op_en (C++ member), 700
esp_ble_mesh_server_recv_gen_move_set_t::tid (C++ member), 700
esp_ble_mesh_server_recv_gen_move_set_t::trans_time (C++ member), 700
esp_ble_mesh_server_recv_gen_onoff_set_t (C++ struct), 698
esp_ble_mesh_server_recv_gen_onoff_set_t::delay (C++ member), 699
esp_ble_mesh_server_recv_gen_onoff_set_t::onoff (C++ member), 699
esp_ble_mesh_server_recv_gen_onoff_set_t::op_en (C++ member), 699
esp_ble_mesh_server_recv_gen_onoff_set_t::tid (C++ member), 699
esp_ble_mesh_server_recv_gen_onoff_set_t::trans_time (C++ member), 699
esp_ble_mesh_server_recv_gen_onpowerup_set_t (C++ struct), 700
esp_ble_mesh_server_recv_gen_onpowerup_set_t::onpowerup (C++ member), 701
esp_ble_mesh_server_recv_gen_power_default_set_t (C++ struct), 701
esp_ble_mesh_server_recv_gen_power_default_set_t::power (C++ member), 701
esp_ble_mesh_server_recv_gen_power_level_set_t (C++ struct), 701
esp_ble_mesh_server_recv_gen_power_level_set_t::delay (C++ member), 701
esp_ble_mesh_server_recv_gen_power_level_set_t::op_en (C++ member), 701
esp_ble_mesh_server_recv_gen_power_level_set_t::power (C++ member), 701
esp_ble_mesh_server_recv_gen_power_level_set_t::tid (C++ member), 701
esp_ble_mesh_server_recv_gen_power_level_set_t::trans_time (C++ member), 701
esp_ble_mesh_server_recv_gen_power_range_set_t (C++ struct), 701
esp_ble_mesh_server_recv_gen_power_range_set_t::range_max (C++ member), 701
esp_ble_mesh_server_recv_gen_power_range_set_t::range_min (C++ member), 701
esp_ble_mesh_server_recv_gen_user_property_set_t (C++ struct), 698
esp_ble_mesh_server_recv_gen_user_property_set_t::property_id (C++ member), 698
esp_ble_mesh_server_recv_light_ctl_default_set_t (C++ struct), 803
esp_ble_mesh_server_recv_light_ctl_default_set_t::ctl_lightness (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_default_set_t::ctl_temperature (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_default_set_t::op_en (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_set_t (C++ struct), 802
esp_ble_mesh_server_recv_light_ctl_set_t::delay (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::delta_uv (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::lightness (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::op_en (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::temperature (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::tid (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_set_t::trans_time (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_temperature_range_set_t (C++ struct), 803
esp_ble_mesh_server_recv_light_ctl_temperature_range_set_t::range_max (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_temperature_range_set_t::range_min (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_temperature_set_t (C++ struct), 802
esp_ble_mesh_server_recv_light_ctl_temperature_set_t::delay (C++ member), 803
esp_ble_mesh_server_recv_light_ctl_temperature_set_t::op_en (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_temperature_set_t::temperature (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_temperature_set_t::tid (C++ member), 802
esp_ble_mesh_server_recv_light_ctl_temperature_set_t::trans_time (C++ member), 802
esp_ble_mesh_server_recv_light_hsl_default_set_t (C++ struct), 805
esp_ble_mesh_server_recv_light_hsl_default_set_t::op_en (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_default_set_t::property_id (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_hue_set_t (C++ struct), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t::delay (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t::hue (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t::op_en (C++ member), 804

esp_ble_mesh_server_recv_light_hsl_hue_set (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t (C++ struct), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t::hue (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_hue_set_t::trans_time (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_range_get (C++ struct), 805
esp_ble_mesh_server_recv_light_hsl_range_set (C++ struct), 805
esp_ble_mesh_server_recv_light_hsl_range_set_t::hue_range_max (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_range_set_t::hue_range_min (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_range_set_t::saturation_range_max (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_range_set_t::saturation_range_min (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_saturation_set (C++ struct), 804
esp_ble_mesh_server_recv_light_hsl_saturation_set_t::delay (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_saturation_set_t::op_en (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_saturation_set_t::resolution (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_saturation_set_t::saturation (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_saturation_set_t::trans_time (C++ member), 805
esp_ble_mesh_server_recv_light_hsl_set (C++ struct), 803
esp_ble_mesh_server_recv_light_hsl_set_t::delay (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_set_t::hue (C++ member), 803
esp_ble_mesh_server_recv_light_hsl_set_t::lightness (C++ member), 803
esp_ble_mesh_server_recv_light_hsl_set_t::op_en (C++ member), 803
esp_ble_mesh_server_recv_light_hsl_set_t::saturation (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_set_t::tid (C++ member), 804
esp_ble_mesh_server_recv_light_hsl_set_t::trans_time (C++ member), 804
esp_ble_mesh_server_recv_light_lc_light_onoff_set (C++ struct), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::delay (C++ member), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::light_onoff (C++ member), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::op_en (C++ member), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::onoff (C++ member), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::tid (C++ member), 807
esp_ble_mesh_server_recv_light_lc_light_onoff_set_t::trans_time (C++ member), 807
esp_ble_mesh_server_recv_light_lc_mode_set (C++ struct), 807
esp_ble_mesh_server_recv_light_lc_mode_set_t::mode (C++ member), 807
esp_ble_mesh_server_recv_light_lc_om_set_t (C++ struct), 807
esp_ble_mesh_server_recv_light_lc_om_set_t::mode (C++ member), 807
esp_ble_mesh_server_recv_light_lc_property_get_t (C++ struct), 800
esp_ble_mesh_server_recv_light_lc_property_get_t::property_id (C++ member), 800
esp_ble_mesh_server_recv_light_lc_property_set_t (C++ struct), 807
esp_ble_mesh_server_recv_light_lc_property_set_t::property_id (C++ member), 808
esp_ble_mesh_server_recv_light_lc_property_set_t::property_value (C++ member), 808
esp_ble_mesh_server_recv_light_lightness_default_set (C++ struct), 801
esp_ble_mesh_server_recv_light_lightness_default_set_t::delay (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_linear_set (C++ struct), 801
esp_ble_mesh_server_recv_light_lightness_linear_set_t::op_en (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_linear_set_t::resolution (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_linear_set_t::tid (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_linear_set_t::trans_time (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_linear_set_t::value (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_range_set (C++ struct), 801
esp_ble_mesh_server_recv_light_lightness_range_set_t::max (C++ member), 802
esp_ble_mesh_server_recv_light_lightness_range_set_t::min (C++ member), 802
esp_ble_mesh_server_recv_light_lightness_set_t (C++ struct), 800
esp_ble_mesh_server_recv_light_lightness_set_t::delay (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_set_t::lightness (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_set_t::op_en (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_set_t::tid (C++ member), 801
esp_ble_mesh_server_recv_light_lightness_set_t::trans_time (C++ member), 801
esp_ble_mesh_server_recv_light_xyl_default_set_t (C++ struct), 806
esp_ble_mesh_server_recv_light_xyl_default_set_t::delay (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_default_set_t::op_en (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_default_set_t::trans_time (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_default_set_t::xyl_default (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_range_set_t (C++ struct), 806

esp_ble_mesh_server_recv_light_xyl_range_set_t::max (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_range_set_t::min (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_range_set_t::max (C++ member), 807
esp_ble_mesh_server_recv_light_xyl_range_set_t::min (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t (C++ struct), 805
esp_ble_mesh_server_recv_light_xyl_set_t::delay (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t::light_mesh (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t::op_en (C++ member), 805
esp_ble_mesh_server_recv_light_xyl_set_t::tid (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t::trans_time (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t::x (C++ member), 806
esp_ble_mesh_server_recv_light_xyl_set_t::y (C++ member), 806
esp_ble_mesh_server_recv_scene_delete_t (C++ struct), 751
esp_ble_mesh_server_recv_scene_delete_t::scene_number (C++ member), 751
esp_ble_mesh_server_recv_scene_recall_t (C++ struct), 751
esp_ble_mesh_server_recv_scene_recall_t::delay (C++ member), 751
esp_ble_mesh_server_recv_scene_recall_t::op_en (C++ member), 751
esp_ble_mesh_server_recv_scene_recall_t::scene_number (C++ member), 751
esp_ble_mesh_server_recv_scene_recall_t::tid (C++ member), 751
esp_ble_mesh_server_recv_scene_recall_t::trans_time (C++ member), 751
esp_ble_mesh_server_recv_scene_store_t (C++ struct), 751
esp_ble_mesh_server_recv_scene_store_t::scene_number (C++ member), 751
esp_ble_mesh_server_recv_scheduler_act_get_t (C++ struct), 749
esp_ble_mesh_server_recv_scheduler_act_get_t::index (C++ member), 749
esp_ble_mesh_server_recv_scheduler_act_set_t (C++ struct), 751
esp_ble_mesh_server_recv_scheduler_act_set_t::action (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::day (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::day_of_week (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::hour (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::index (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::max (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::min (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::month (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::scene_number (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::second (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::trans_time (C++ member), 752
esp_ble_mesh_server_recv_scheduler_act_set_t::year (C++ member), 752
esp_ble_mesh_server_recv_sensor_cadence_get_t (C++ struct), 723
esp_ble_mesh_server_recv_sensor_cadence_get_t::property_id (C++ member), 723
esp_ble_mesh_server_recv_sensor_cadence_set_t (C++ struct), 724
esp_ble_mesh_server_recv_sensor_cadence_set_t::fast_cadence_high (C++ member), 725
esp_ble_mesh_server_recv_sensor_cadence_set_t::fast_cadence_low (C++ member), 725
esp_ble_mesh_server_recv_sensor_column_get_t (C++ struct), 724
esp_ble_mesh_server_recv_sensor_column_get_t::property_id (C++ member), 724
esp_ble_mesh_server_recv_sensor_column_get_t::raw_value_x (C++ member), 724
esp_ble_mesh_server_recv_sensor_descriptor_get_t (C++ struct), 723
esp_ble_mesh_server_recv_sensor_descriptor_get_t::op_en (C++ member), 723
esp_ble_mesh_server_recv_sensor_descriptor_get_t::property_id (C++ member), 723
esp_ble_mesh_server_recv_sensor_get_t (C++ struct), 724
esp_ble_mesh_server_recv_sensor_get_t::op_en (C++ member), 724
esp_ble_mesh_server_recv_sensor_get_t::property_id (C++ member), 724
esp_ble_mesh_server_recv_sensor_series_get_t (C++ struct), 724
esp_ble_mesh_server_recv_sensor_series_get_t::op_en (C++ member), 724
esp_ble_mesh_server_recv_sensor_series_get_t::property_id (C++ member), 724
esp_ble_mesh_server_recv_sensor_series_get_t::raw_value_x1 (C++ member), 724
esp_ble_mesh_server_recv_sensor_setting_get_t (C++ struct), 723
esp_ble_mesh_server_recv_sensor_setting_get_t::property_id (C++ member), 724
esp_ble_mesh_server_recv_sensor_setting_get_t::setting_id (C++ member), 724
esp_ble_mesh_server_recv_sensor_setting_set_t (C++ struct), 725

esp_ble_mesh_server_recv_sensor_setting_get_t::property_id (C++ member), 725
esp_ble_mesh_server_recv_sensor_setting_set_t (C++ struct), 725
esp_ble_mesh_server_recv_sensor_setting_set_t::property_id (C++ member), 725
esp_ble_mesh_server_recv_sensor_setting_set_t::sensor_raw (C++ member), 725
esp_ble_mesh_server_recv_sensor_status_get_t (C++ struct), 723
esp_ble_mesh_server_recv_sensor_status_get_t::property_id (C++ member), 723
esp_ble_mesh_server_recv_sensor_status_set_t (C++ struct), 808
esp_ble_mesh_server_recv_sensor_status_set_t::property_id (C++ member), 808
esp_ble_mesh_server_recv_tai_utc_delta_get_t (C++ struct), 750
esp_ble_mesh_server_recv_tai_utc_delta_set_t (C++ struct), 750
esp_ble_mesh_server_recv_tai_utc_delta_set_t::padding (C++ member), 750
esp_ble_mesh_server_recv_tai_utc_delta_set_t::tai_utc_delta (C++ member), 750
esp_ble_mesh_server_recv_tai_utc_delta_set_t::time_authority (C++ member), 750
esp_ble_mesh_server_recv_time_role_set_t (C++ struct), 750
esp_ble_mesh_server_recv_time_role_set_t::time_role (C++ member), 751
esp_ble_mesh_server_recv_time_set_t (C++ struct), 749
esp_ble_mesh_server_recv_time_set_t::subsecond (C++ member), 750
esp_ble_mesh_server_recv_time_set_t::tai_seconds (C++ member), 750
esp_ble_mesh_server_recv_time_set_t::tai_utc_delta (C++ member), 750
esp_ble_mesh_server_recv_time_set_t::time_authority (C++ member), 750
esp_ble_mesh_server_recv_time_set_t::time_zone_offset (C++ member), 750
esp_ble_mesh_server_recv_time_set_t::uncertainty (C++ member), 750
esp_ble_mesh_server_recv_time_status_t (C++ struct), 752
esp_ble_mesh_server_recv_time_status_t::subsecond (C++ member), 752
esp_ble_mesh_server_recv_time_status_t::tai_seconds (C++ member), 752
esp_ble_mesh_server_recv_time_status_t::tai_utc_delta (C++ member), 753
esp_ble_mesh_server_recv_time_status_t::time_authority (C++ member), 752
esp_ble_mesh_server_recv_time_status_t::time_zone_offset (C++ member), 753
esp_ble_mesh_server_recv_time_status_t::uncertainty (C++ member), 752
esp_ble_mesh_server_recv_time_zone_set_t (C++ struct), 750
esp_ble_mesh_server_recv_time_zone_set_t::time_zone_offset (C++ member), 750
esp_ble_mesh_server_recv_time_zone_set_t::time_zone_offset_new (C++ member), 750
ESP_BLE_MESH_SERVER_RECV_TIME_ZONE_SET_EVT (C++ enumerator), 606
ESP_BLE_MESH_SERVER_RSP_BY_APP (C macro), 596
esp_ble_mesh_server_rsp_ctrl_t (C++ struct), 573
esp_ble_mesh_server_rsp_ctrl_t::get_auto_rsp (C++ member), 573
esp_ble_mesh_server_rsp_ctrl_t::set_auto_rsp (C++ member), 574
esp_ble_mesh_server_rsp_ctrl_t::status_auto_rsp (C++ member), 574
esp_ble_mesh_server_state_type_t (C++ enum), 606
ESP_BLE_MESH_GEN_BATTERY_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_DEFAULT_TRANS_TIME_SERVER (C++ enumerator), 606
ESP_BLE_MESH_GEN_DELTA_LEVEL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_DTT_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_LEVEL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_LOCATION_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_MANUFACTURER_PROPERTIES_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_ONOFF_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_ONPOWERUP_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_POWER_LEVEL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_POWER_ONOFF_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_POWER_ONOFF_SETUP_SERVER (C++ enumerator), 607
ESP_BLE_MESH_GEN_PROPERTY_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_CTL_HUE_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_CTL_SATURATION_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_CTL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_CTL_SETUP_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_HSL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_HSL_SETUP_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_LC_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_LIGHTNESS_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_XYL_SERVER (C++ enumerator), 607
ESP_BLE_MESH_LIGHT_XYL_SETUP_SERVER (C++ enumerator), 607
ESP_BLE_MESH_SENSOR_SERVER (C++ enumerator), 607
esp_ble_mesh_server_state_value_t (C++ union), 558
esp_ble_mesh_server_state_value_t::delta_uv (C++ member), 560
esp_ble_mesh_server_state_value_t::gen_level (C++ member), 559
esp_ble_mesh_server_state_value_t::gen_onoff (C++ member), 559
esp_ble_mesh_server_state_value_t::gen_onpowerup (C++ member), 559
esp_ble_mesh_server_state_value_t::gen_power_actual (C++ member), 559
esp_ble_mesh_server_state_value_t::hue (C++ member), 560

esp_ble_mesh_server_state_value_t::level (C++ member), 559
esp_ble_mesh_server_state_value_t::light_ctl_temp (C++ member), 559
esp_ble_mesh_server_state_value_t::light_ctl_uv (C++ member), 560
esp_ble_mesh_server_state_value_t::light_hsl (C++ member), 560
esp_ble_mesh_server_state_value_t::light_hsl_hue (C++ member), 560
esp_ble_mesh_server_state_value_t::light_hsl_saturation (C++ member), 560
esp_ble_mesh_server_state_value_t::light_lc_light_onoff (C++ member), 560
esp_ble_mesh_server_state_value_t::light_lightness_actual (C++ member), 559
esp_ble_mesh_server_state_value_t::light_lightness_linear (C++ member), 559
esp_ble_mesh_server_state_value_t::light_xyl_x (C++ member), 560
esp_ble_mesh_server_state_value_t::lightness (C++ member), 559
esp_ble_mesh_server_state_value_t::onoff (C++ member), 559
esp_ble_mesh_server_state_value_t::onpowerup (C++ member), 559
esp_ble_mesh_server_state_value_t::power (C++ member), 559
esp_ble_mesh_server_state_value_t::saturation (C++ member), 560
esp_ble_mesh_server_state_value_t::temperature (C++ member), 559
esp_ble_mesh_set_fast_prov_action (C++ function), 625
esp_ble_mesh_set_fast_prov_info (C++ function), 625
esp_ble_mesh_set_unprovisioned_device_name (C++ function), 621
ESP_BLE_MESH_SETTINGS_UID_SIZE (C macro), 574
ESP_BLE_MESH_SIG_MODEL (C macro), 577
esp_ble_mesh_state_change_cfg_appkey_add_t (C++ struct), 655
esp_ble_mesh_state_change_cfg_appkey_add_t::app_idx (C++ member), 655
esp_ble_mesh_state_change_cfg_appkey_add_t::appkey (C++ member), 655
esp_ble_mesh_state_change_cfg_appkey_add_t::net_idx (C++ member), 655
esp_ble_mesh_state_change_cfg_appkey_delete_t (C++ struct), 656
esp_ble_mesh_state_change_cfg_appkey_delete_t::app_idx (C++ member), 656
esp_ble_mesh_state_change_cfg_appkey_delete_t::net_idx (C++ member), 656
esp_ble_mesh_state_change_cfg_appkey_update_t (C++ struct), 656
esp_ble_mesh_state_change_cfg_appkey_update_t::app_idx (C++ member), 656
esp_ble_mesh_state_change_cfg_appkey_update_t::appkey (C++ member), 656
esp_ble_mesh_state_change_cfg_appkey_update_t::net_idx (C++ member), 656
esp_ble_mesh_state_change_cfg_kr_phase_set_t (C++ struct), 657
esp_ble_mesh_state_change_cfg_kr_phase_set_t::kr_phase (C++ member), 657
esp_ble_mesh_state_change_cfg_kr_phase_set_t::net_idx (C++ member), 657
esp_ble_mesh_state_change_cfg_mod_pub_set_t (C++ struct), 653
esp_ble_mesh_state_change_cfg_mod_pub_set_t::app_idx (C++ member), 653
esp_ble_mesh_state_change_cfg_mod_pub_set_t::company_id (C++ member), 654
esp_ble_mesh_state_change_cfg_mod_pub_set_t::cred_flag (C++ member), 653
esp_ble_mesh_state_change_cfg_mod_pub_set_t::element_addr (C++ member), 653
esp_ble_mesh_state_change_cfg_mod_pub_set_t::model_id (C++ member), 654
esp_ble_mesh_state_change_cfg_mod_pub_set_t::publish_addr (C++ member), 654
esp_ble_mesh_state_change_cfg_mod_pub_set_t::publish_period (C++ member), 654
esp_ble_mesh_state_change_cfg_mod_pub_set_t::publish_retransmit (C++ member), 654
esp_ble_mesh_state_change_cfg_mod_pub_set_t::publish_ttl (C++ member), 654
esp_ble_mesh_state_change_cfg_model_app_bind_t (C++ struct), 656
esp_ble_mesh_state_change_cfg_model_app_bind_t::app_idx (C++ member), 656
esp_ble_mesh_state_change_cfg_model_app_bind_t::company_id (C++ member), 656
esp_ble_mesh_state_change_cfg_model_app_bind_t::element_addr (C++ member), 656
esp_ble_mesh_state_change_cfg_model_app_bind_t::model_id (C++ member), 656
esp_ble_mesh_state_change_cfg_model_app_unbind_t (C++ struct), 656
esp_ble_mesh_state_change_cfg_model_app_unbind_t::app_idx (C++ member), 657
esp_ble_mesh_state_change_cfg_model_app_unbind_t::company_id (C++ member), 657
esp_ble_mesh_state_change_cfg_model_app_unbind_t::element_addr (C++ member), 657
esp_ble_mesh_state_change_cfg_model_app_unbind_t::model_id (C++ member), 657
esp_ble_mesh_state_change_cfg_model_sub_add_t (C++ struct), 654
esp_ble_mesh_state_change_cfg_model_sub_add_t::company_id (C++ member), 654
esp_ble_mesh_state_change_cfg_model_sub_add_t::el

(*C++ member*), 654
esp_ble_mesh_state_change_cfg_model_sub_add_t::model_id
(*C++ member*), 654
esp_ble_mesh_state_change_cfg_model_sub_add_t::sub_addr
(*C++ member*), 654
esp_ble_mesh_state_change_cfg_model_sub_delete_t
(*C++ struct*), 654
esp_ble_mesh_state_change_cfg_model_sub_delete_t::company_id
(*C++ member*), 654
esp_ble_mesh_state_change_cfg_model_sub_delete_t::element_addr
(*C++ member*), 654
esp_ble_mesh_state_change_cfg_model_sub_delete_t::model_id
(*C++ member*), 655
esp_ble_mesh_state_change_cfg_model_sub_delete_t::sub_addr
(*C++ member*), 654
esp_ble_mesh_state_change_cfg_netkey_add_t
(*C++ struct*), 655
esp_ble_mesh_state_change_cfg_netkey_add_t::net_idx
(*C++ member*), 655
esp_ble_mesh_state_change_cfg_netkey_add_t::net_key
(*C++ member*), 655
esp_ble_mesh_state_change_cfg_netkey_delete_t
(*C++ struct*), 655
esp_ble_mesh_state_change_cfg_netkey_delete_t::net_idx
(*C++ member*), 655
esp_ble_mesh_state_change_cfg_netkey_update_t
(*C++ struct*), 655
esp_ble_mesh_state_change_cfg_netkey_update_t::net_idx
(*C++ member*), 655
esp_ble_mesh_state_change_cfg_netkey_update_t::net_key
(*C++ member*), 655
esp_ble_mesh_state_change_gen_admin_property_set_t
(*C++ struct*), 697
esp_ble_mesh_state_change_gen_admin_property_set_t::access
(*C++ member*), 697
esp_ble_mesh_state_change_gen_admin_property_set_t::id
(*C++ member*), 697
esp_ble_mesh_state_change_gen_admin_property_set_t::value
(*C++ member*), 697
esp_ble_mesh_state_change_gen_def_trans_time_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_def_trans_time_set_t::trans_time
(*C++ member*), 695
esp_ble_mesh_state_change_gen_delta_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_delta_set_t::level
(*C++ member*), 695
esp_ble_mesh_state_change_gen_level_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_level_set_t::level
(*C++ member*), 695
esp_ble_mesh_state_change_gen_loc_global_set_t
(*C++ struct*), 696
esp_ble_mesh_state_change_gen_loc_global_set_t::altitude
(*C++ member*), 696
esp_ble_mesh_state_change_gen_loc_global_set_t::latitude
(*C++ member*), 696
esp_ble_mesh_state_change_gen_loc_global_set_t::longitude
(*C++ member*), 696
esp_ble_mesh_state_change_gen_loc_local_set_t
(*C++ struct*), 697
esp_ble_mesh_state_change_gen_loc_local_set_t::al
(*C++ member*), 697
esp_ble_mesh_state_change_gen_loc_local_set_t::ea
(*C++ member*), 697
esp_ble_mesh_state_change_gen_loc_local_set_t::fl
(*C++ member*), 697
esp_ble_mesh_state_change_gen_loc_local_set_t::no
(*C++ member*), 697
esp_ble_mesh_state_change_gen_loc_local_set_t::un
(*C++ member*), 697
esp_ble_mesh_state_change_gen_manu_property_set_t
(*C++ struct*), 698
esp_ble_mesh_state_change_gen_manu_property_set_t
(*C++ member*), 698
esp_ble_mesh_state_change_gen_manu_property_set_t
(*C++ member*), 698
esp_ble_mesh_state_change_gen_move_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_move_set_t::level
(*C++ member*), 695
esp_ble_mesh_state_change_gen_onoff_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_onoff_set_t::onoff
(*C++ member*), 695
esp_ble_mesh_state_change_gen_onpowerup_set_t
(*C++ struct*), 695
esp_ble_mesh_state_change_gen_onpowerup_set_t::on
(*C++ member*), 696
esp_ble_mesh_state_change_gen_power_default_set_t
(*C++ struct*), 696
esp_ble_mesh_state_change_gen_power_default_set_t
(*C++ member*), 696
esp_ble_mesh_state_change_gen_power_level_set_t
(*C++ struct*), 696
esp_ble_mesh_state_change_gen_power_level_set_t::
(*C++ member*), 696
esp_ble_mesh_state_change_gen_power_range_set_t
(*C++ struct*), 696
esp_ble_mesh_state_change_gen_power_range_set_t::
(*C++ member*), 696
esp_ble_mesh_state_change_gen_power_range_set_t::
(*C++ member*), 696
esp_ble_mesh_state_change_gen_user_property_set_t
(*C++ struct*), 697
esp_ble_mesh_state_change_gen_user_property_set_t
(*C++ member*), 697
esp_ble_mesh_state_change_gen_user_property_set_t
(*C++ member*), 697
esp_ble_mesh_state_change_light_ctl_default_set_t
(*C++ struct*), 796
esp_ble_mesh_state_change_light_ctl_default_set_t
(*C++ member*), 797
esp_ble_mesh_state_change_light_ctl_default_set_t
(*C++ member*), 797
esp_ble_mesh_state_change_light_ctl_default_set_t

(C++ member), 797
esp_ble_mesh_state_change_light_ctl_set_t
(C++ struct), 796
esp_ble_mesh_state_change_light_ctl_set_t::ctl
(C++ member), 796
esp_ble_mesh_state_change_light_ctl_set_t::lightness
(C++ member), 796
esp_ble_mesh_state_change_light_ctl_set_t::temperature
(C++ member), 796
esp_ble_mesh_state_change_light_ctl_temp_set_t
(C++ struct), 796
esp_ble_mesh_state_change_light_ctl_temp_set_t::delta_uv
(C++ member), 796
esp_ble_mesh_state_change_light_ctl_temp_set_t::temperature
(C++ member), 796
esp_ble_mesh_state_change_light_hsl_default_set_t
(C++ struct), 797
esp_ble_mesh_state_change_light_hsl_default_set_t::lightness
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_default_set_t::saturation
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_hue_set_t
(C++ struct), 797
esp_ble_mesh_state_change_light_hsl_hue_set_t::hue
(C++ member), 797
esp_ble_mesh_state_change_light_hsl_range_set_t
(C++ struct), 798
esp_ble_mesh_state_change_light_hsl_range_set_t::hue_range_max
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_range_set_t::hue_range_min
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_range_set_t::saturation_range_max
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_range_set_t::saturation_range_min
(C++ member), 798
esp_ble_mesh_state_change_light_hsl_sat_set_t
(C++ struct), 797
esp_ble_mesh_state_change_light_hsl_sat_set_t::saturation
(C++ member), 797
esp_ble_mesh_state_change_light_hsl_set_t
(C++ struct), 797
esp_ble_mesh_state_change_light_hsl_set_t::hue
(C++ member), 797
esp_ble_mesh_state_change_light_hsl_set_t::lightness
(C++ member), 797
esp_ble_mesh_state_change_light_hsl_set_t::saturation
(C++ member), 797
esp_ble_mesh_state_change_light_lc_light_onoff_set_t
(C++ struct), 799
esp_ble_mesh_state_change_light_lc_light_onoff_set_t::onoff
(C++ member), 800
esp_ble_mesh_state_change_light_lc_mode_set_t
(C++ struct), 799
esp_ble_mesh_state_change_light_lc_mode_set_t::mode
(C++ member), 799
esp_ble_mesh_state_change_light_lc_om_set_t
(C++ struct), 799
esp_ble_mesh_state_change_light_lc_om_set_t::mode
(C++ member), 799
esp_ble_mesh_state_change_light_lc_property_set_t
(C++ struct), 800
esp_ble_mesh_state_change_light_lc_property_set_t::property_id
(C++ member), 800
esp_ble_mesh_state_change_light_lc_property_set_t::property_value
(C++ member), 800
esp_ble_mesh_state_change_light_lightness_default_set_t
(C++ struct), 795
esp_ble_mesh_state_change_light_lightness_default_set_t::default_lightness
(C++ member), 795
esp_ble_mesh_state_change_light_lightness_linear_set_t
(C++ struct), 795
esp_ble_mesh_state_change_light_lightness_linear_set_t::lightness
(C++ member), 795
esp_ble_mesh_state_change_light_lightness_range_set_t
(C++ struct), 795
esp_ble_mesh_state_change_light_lightness_range_set_t::max_lightness
(C++ member), 796
esp_ble_mesh_state_change_light_lightness_range_set_t::min_lightness
(C++ member), 796
esp_ble_mesh_state_change_light_lightness_set_t
(C++ struct), 795
esp_ble_mesh_state_change_light_lightness_set_t::lightness
(C++ member), 795
esp_ble_mesh_state_change_light_xyl_default_set_t
(C++ struct), 798
esp_ble_mesh_state_change_light_xyl_default_set_t::default_intensity
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_default_set_t::default_chromaticity
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_default_set_t::range_change_rule
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_range_set_t
(C++ struct), 799
esp_ble_mesh_state_change_light_xyl_range_set_t::color_range_max
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_range_set_t::color_range_min
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_range_set_t::intensity_range_max
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_range_set_t::intensity_range_min
(C++ member), 799
esp_ble_mesh_state_change_light_xyl_set_t
(C++ struct), 798
esp_ble_mesh_state_change_light_xyl_set_t::lightness
(C++ member), 798
esp_ble_mesh_state_change_light_xyl_set_t::x
(C++ member), 798
esp_ble_mesh_state_change_light_xyl_set_t::y

(C++ member), 798
esp_ble_mesh_state_change_scene_delete_t
(C++ struct), 748
esp_ble_mesh_state_change_scene_delete_t::scene_number
(C++ member), 748
esp_ble_mesh_state_change_scene_recall_t
(C++ struct), 748
esp_ble_mesh_state_change_scene_recall_t::scene_number
(C++ member), 748
esp_ble_mesh_state_change_scene_store_t
(C++ struct), 748
esp_ble_mesh_state_change_scene_store_t::scene_number
(C++ member), 748
esp_ble_mesh_state_change_scheduler_act_set_t
(C++ struct), 748
esp_ble_mesh_state_change_scheduler_act_set_t::action
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::day
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::day_of_week
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::hour
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::index
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::minute
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::month
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::scene_number
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::second
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::trans_time
(C++ member), 749
esp_ble_mesh_state_change_scheduler_act_set_t::year
(C++ member), 749
esp_ble_mesh_state_change_sensor_cadence_set_t
(C++ struct), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::fast_cadence_high
(C++ member), 723
esp_ble_mesh_state_change_sensor_cadence_set_t::fast_cadence_low
(C++ member), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::min_interval
(C++ member), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::period_divisor
(C++ member), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::trigger_delta_down
(C++ member), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::trigger_delta_up
(C++ member), 722
esp_ble_mesh_state_change_sensor_cadence_set_t::trigger_type
(C++ member), 722
esp_ble_mesh_state_change_sensor_setting_set_t
(C++ struct), 723
esp_ble_mesh_state_change_sensor_setting_set_t::property_id
(C++ member), 723
esp_ble_mesh_state_change_sensor_setting_set_t::sensor_setting_raw
(C++ member), 723
esp_ble_mesh_state_change_sensor_status_t
(C++ struct), 800
esp_ble_mesh_state_change_sensor_status_t::ambient_set
(C++ member), 800
esp_ble_mesh_state_change_sensor_status_t::occupancy
(C++ member), 800
esp_ble_mesh_state_change_sensor_status_t::property_id
(C++ member), 800
esp_ble_mesh_state_change_sensor_status_t::set_occupancy
(C++ member), 800
esp_ble_mesh_state_change_sensor_status_t::state
(C++ member), 800
esp_ble_mesh_state_change_tai_utc_delta_set_t
(C++ struct), 748
esp_ble_mesh_state_change_tai_utc_delta_set_t::tai_utc_delta_new
(C++ member), 748
esp_ble_mesh_state_change_tai_utc_delta_set_t::tai_utc_delta_old
(C++ member), 748
esp_ble_mesh_state_change_time_role_set_t
(C++ struct), 748
esp_ble_mesh_state_change_time_role_set_t::time_role
(C++ member), 748
esp_ble_mesh_state_change_time_set_t
(C++ struct), 746
esp_ble_mesh_state_change_time_set_t::subsecond
(C++ member), 746
esp_ble_mesh_state_change_time_set_t::tai_seconds
(C++ member), 746
esp_ble_mesh_state_change_time_set_t::tai_utc_delta
(C++ member), 747
esp_ble_mesh_state_change_time_set_t::time_authority
(C++ member), 747
esp_ble_mesh_state_change_time_set_t::time_zone_offset_new
(C++ member), 747
esp_ble_mesh_state_change_time_set_t::uncertainty
(C++ member), 747
esp_ble_mesh_state_change_time_status_t
(C++ struct), 747
esp_ble_mesh_state_change_time_status_t::subsecond
(C++ member), 747
esp_ble_mesh_state_change_time_status_t::tai_seconds
(C++ member), 747
esp_ble_mesh_state_change_time_status_t::tai_utc_delta
(C++ member), 747
esp_ble_mesh_state_change_time_status_t::time_authority
(C++ member), 747
esp_ble_mesh_state_change_time_status_t::time_zone_offset
(C++ member), 747
esp_ble_mesh_state_change_time_zone_set_t
(C++ struct), 747
esp_ble_mesh_state_change_time_zone_set_t::tai_zone_change

(C++ member), 747
esp_ble_mesh_state_change_time_zone_set_t::time_zone_offset_new (C++ member), 747
esp_ble_mesh_state_transition_t (C++ struct), 572
esp_ble_mesh_state_transition_t::BLE_MESH_ATOMIC_DEFINE (C++ function), 572
esp_ble_mesh_state_transition_t::counter (C++ member), 573
esp_ble_mesh_state_transition_t::delay (C++ member), 572
esp_ble_mesh_state_transition_t::just_started (C++ member), 572
esp_ble_mesh_state_transition_t::quo_tt (C++ member), 573
esp_ble_mesh_state_transition_t::remain_time (C++ member), 572
esp_ble_mesh_state_transition_t::start_timestamp (C++ member), 573
esp_ble_mesh_state_transition_t::timer (C++ member), 573
esp_ble_mesh_state_transition_t::total_duration (C++ member), 573
esp_ble_mesh_state_transition_t::trans_time (C++ member), 572
ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_ERROR (C macro), 667
ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_WARNING (C macro), 667
ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_ERROR (C macro), 667
ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_WARNING (C macro), 667
ESP_BLE_MESH_TAI_OF_DELTA_CHANGE_LEN (C macro), 756
ESP_BLE_MESH_TAI_OF_ZONE_CHANGE_LEN (C macro), 756
ESP_BLE_MESH_TAI_SECONDS_LEN (C macro), 756
ESP_BLE_MESH_TAI_UTC_DELTA_MAX_VALUE (C macro), 755
esp_ble_mesh_tai_utc_delta_set_t (C++ struct), 735
esp_ble_mesh_tai_utc_delta_set_t::padding (C++ member), 735
esp_ble_mesh_tai_utc_delta_set_t::tai_utc_delta_new (C++ member), 736
esp_ble_mesh_tai_utc_delta_set_t::tai_utc_delta_new (C++ member), 735
esp_ble_mesh_tai_utc_delta_status_cb_t (C++ struct), 738
esp_ble_mesh_tai_utc_delta_status_cb_t::padding_1 (C++ member), 739
esp_ble_mesh_tai_utc_delta_status_cb_t::padding_2 (C++ member), 739
esp_ble_mesh_tai_utc_delta_status_cb_t::tai_utc_delta_change (C++ member), 739
esp_ble_mesh_tai_utc_delta_status_cb_t::tai_utc_delta_timer (C++ member), 739
esp_ble_mesh_tai_utc_delta_status_cb_t::tai_utc_delta_status_cb_t::tai_utc_delta_new (C++ member), 739
ESP_BLE_MESH_TAMPER_ERROR (C macro), 668
ESP_BLE_MESH_TAMPER_WARNING (C macro), 668
ESP_BLE_MESH_TIME_AUTHORITY (C macro), 757
ESP_BLE_MESH_TIME_CLINET (C macro), 757
ESP_BLE_MESH_TIME_NONE (C macro), 757
ESP_BLE_MESH_TIME_RELAY (C macro), 757
esp_ble_mesh_time_role_set_t (C++ struct), 736
esp_ble_mesh_time_role_set_t::time_role (C++ member), 736
esp_ble_mesh_time_role_status_cb_t (C++ struct), 739
esp_ble_mesh_time_role_status_cb_t::time_role (C++ member), 739
esp_ble_mesh_time_scene_client_cb_event_t (C++ enum), 758
esp_ble_mesh_time_scene_client_cb_event_t::ESP_BLE_MESH_TIME_SCENE_CLIENT_GET_STATE_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_client_cb_event_t::ESP_BLE_MESH_TIME_SCENE_CLIENT_PUBLISH_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_client_cb_event_t::ESP_BLE_MESH_TIME_SCENE_CLIENT_SET_STATE_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_client_cb_event_t::ESP_BLE_MESH_TIME_SCENE_CLIENT_TIMEOUT_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_client_cb_event_t::ESP_BLE_MESH_TIME_SCENE_EVT_MAX (C++ enumerator), 758
esp_ble_mesh_time_scene_client_cb_param_t (C++ struct), 741
esp_ble_mesh_time_scene_client_cb_param_t::error_code (C++ member), 741
esp_ble_mesh_time_scene_client_cb_param_t::params (C++ member), 741
esp_ble_mesh_time_scene_client_cb_param_t::status_cb (C++ member), 741
esp_ble_mesh_time_scene_client_cb_t (C++ type), 757
esp_ble_mesh_time_scene_client_get_state (C++ function), 730
esp_ble_mesh_time_scene_client_get_state_t (C++ union), 731
esp_ble_mesh_time_scene_client_get_state_t::sched_get (C++ member), 731
esp_ble_mesh_time_scene_client_set_state (C++ function), 731
esp_ble_mesh_time_scene_client_set_state_t (C++ union), 731
esp_ble_mesh_time_scene_client_set_state_t::scene_set (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::scene_set_unack (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::sched_set (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::sched_set_unack (C++ member), 732

(C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::time_scene (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::time_scene_id (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::time_scene_server_recv_get_msg_t (C++ member), 732
esp_ble_mesh_time_scene_client_set_state_t::time_scene_server_recv_set_msg_t (C++ member), 731
esp_ble_mesh_time_scene_client_set_state_t::time_scene_server_recv_status_msg_t (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t (C++ union), 732
esp_ble_mesh_time_scene_client_status_cb_t::scene_register_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::scene_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::scheduler_action_status (C++ member), 733
esp_ble_mesh_time_scene_client_status_cb_t::scheduler_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::time_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::time_zone_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::tai_utc_delta_status (C++ member), 732
esp_ble_mesh_time_scene_client_status_cb_t::time_role_status (C++ member), 732
esp_ble_mesh_time_scene_message_opcode_t (C++ type), 598
esp_ble_mesh_time_scene_server_cb_event_t (C++ enum), 758
esp_ble_mesh_time_scene_server_cb_event_t::ESP_BLE_MESH_TIME_SCENE_SERVER_EVT_MAX (C++ enumerator), 759
esp_ble_mesh_time_scene_server_cb_event_t::ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_GET_MSG_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_server_cb_event_t::ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_SET_MSG_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_server_cb_event_t::ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_STATUS_MSG_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_server_cb_event_t::ESP_BLE_MESH_TIME_SCENE_SERVER_STATE_CHANGE_EVT (C++ enumerator), 758
esp_ble_mesh_time_scene_server_cb_param_t (C++ struct), 753
esp_ble_mesh_time_scene_server_cb_param_t::ctx (C++ member), 753
esp_ble_mesh_time_scene_server_cb_param_t::model (C++ member), 753
esp_ble_mesh_time_scene_server_cb_param_t::value (C++ member), 753
esp_ble_mesh_time_scene_server_cb_t (C++ type), 758
esp_ble_mesh_time_scene_server_cb_value_t (C++ union), 734
esp_ble_mesh_time_scene_server_cb_value_t::get (C++ member), 734
esp_ble_mesh_time_scene_server_cb_value_t::set (C++ member), 734
esp_ble_mesh_time_scene_server_cb_value_t::status (C++ member), 734
esp_ble_mesh_time_scene_server_recv_get_msg_t (C++ union), 733
esp_ble_mesh_time_scene_server_recv_get_msg_t::scene (C++ member), 733
esp_ble_mesh_time_scene_server_recv_set_msg_t (C++ union), 733
esp_ble_mesh_time_scene_server_recv_set_msg_t::scene (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::scene_register (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::scene_store (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::scheduler_action (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::time (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::time_role (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::time_zone_offset_new (C++ member), 734
esp_ble_mesh_time_scene_server_recv_set_msg_t::tai_utc_delta_new (C++ member), 734
esp_ble_mesh_time_scene_server_recv_status_msg_t (C++ union), 734
esp_ble_mesh_time_scene_server_recv_status_msg_t::time_zone_offset_status (C++ member), 734
esp_ble_mesh_time_scene_server_state_change_t (C++ union), 733
esp_ble_mesh_time_scene_server_state_change_t::scene_recall (C++ member), 733
esp_ble_mesh_time_scene_server_state_change_t::scene_register_get (C++ member), 733
esp_ble_mesh_time_scene_server_state_change_t::scene_store (C++ member), 733
esp_ble_mesh_time_scene_server_state_change_t::time_role (C++ member), 733
esp_ble_mesh_time_set_t (C++ struct), 735
esp_ble_mesh_time_set_t::sub_second (C++ member), 735
esp_ble_mesh_time_set_t::tai_seconds (C++ member), 735
esp_ble_mesh_time_set_t::tai_utc_delta (C++ member), 735
esp_ble_mesh_time_set_t::time_authority (C++ member), 735

esp_ble_mesh_time_set_t::time_zone_offset_new (C++ member), 735
esp_ble_mesh_time_set_t::uncertainty (C++ member), 735
esp_ble_mesh_time_setup_srv_t (C++ struct), 742
esp_ble_mesh_time_setup_srv_t::model (C++ member), 742
esp_ble_mesh_time_setup_srv_t::rsp_ctrl (C++ member), 742
esp_ble_mesh_time_setup_srv_t::state (C++ member), 742
esp_ble_mesh_time_srv_t (C++ struct), 742
esp_ble_mesh_time_srv_t::model (C++ member), 742
esp_ble_mesh_time_srv_t::rsp_ctrl (C++ member), 742
esp_ble_mesh_time_srv_t::state (C++ member), 742
esp_ble_mesh_time_state_t (C++ struct), 741
esp_ble_mesh_time_state_t::subsecond (C++ member), 741
esp_ble_mesh_time_state_t::tai_delta_change (C++ member), 742
esp_ble_mesh_time_state_t::tai_seconds (C++ member), 741
esp_ble_mesh_time_state_t::tai_utc_delta_curr (C++ member), 742
esp_ble_mesh_time_state_t::tai_utc_delta_new (C++ member), 742
esp_ble_mesh_time_state_t::tai_zone_change (C++ member), 741
esp_ble_mesh_time_state_t::time (C++ member), 742
esp_ble_mesh_time_state_t::time_authority (C++ member), 741
esp_ble_mesh_time_state_t::time_role (C++ member), 742
esp_ble_mesh_time_state_t::time_zone_offset_curr (C++ member), 741
esp_ble_mesh_time_state_t::time_zone_offset_new (C++ member), 741
esp_ble_mesh_time_state_t::uncertainty (C++ member), 741
esp_ble_mesh_time_status_cb_t (C++ struct), 738
esp_ble_mesh_time_status_cb_t::sub_second (C++ member), 738
esp_ble_mesh_time_status_cb_t::tai_seconds (C++ member), 738
esp_ble_mesh_time_status_cb_t::tai_utc_delta (C++ member), 738
esp_ble_mesh_time_status_cb_t::time_authority (C++ member), 738
esp_ble_mesh_time_status_cb_t::time_zone_offset (C++ member), 738
esp_ble_mesh_time_status_cb_t::uncertainty (C++ member), 738
esp_ble_mesh_time_zone_set_t (C++ struct), 735
esp_ble_mesh_time_zone_set_t::tai_zone_change (C++ member), 735
esp_ble_mesh_time_zone_set_t::time_zone_offset_new (C++ member), 735
esp_ble_mesh_time_zone_status_cb_t (C++ struct), 738
esp_ble_mesh_time_zone_status_cb_t::tai_zone_change (C++ member), 738
esp_ble_mesh_time_zone_status_cb_t::time_zone_offset_curr (C++ member), 738
esp_ble_mesh_time_zone_status_cb_t::time_zone_offset_new (C++ member), 738
ESP_BLE_MESH_TRANSMIT (C macro), 576
ESP_BLE_MESH_TTL_DEFAULT (C macro), 574
ESP_BLE_MESH_TTL_MAX (C macro), 574
ESP_BLE_MESH_UNKNOWN_TAI_DELTA_CHANGE (C macro), 755
ESP_BLE_MESH_UNKNOWN_TAI_SECONDS (C macro), 755
ESP_BLE_MESH_UNKNOWN_TAI_ZONE_CHANGE (C macro), 755
esp_ble_mesh_unprov_dev_add_t (C++ struct), 568
esp_ble_mesh_unprov_dev_add_t::addr (C++ member), 568
esp_ble_mesh_unprov_dev_add_t::addr_type (C++ member), 568
esp_ble_mesh_unprov_dev_add_t::bearer (C++ member), 568
esp_ble_mesh_unprov_dev_add_t::oob_info (C++ member), 568
esp_ble_mesh_unprov_dev_add_t::uuid (C++ member), 568
ESP_BLE_MESH_VENDOR_MODEL (C macro), 577
ESP_BLE_MESH_VIBRATION_ERROR (C macro), 667
ESP_BLE_MESH_VIBRATION_WARNING (C macro), 667
ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE (C macro), 294
ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE (C macro), 294
ESP_BLE_OOB_DISABLE (C macro), 295
ESP_BLE_OOB_ENABLE (C macro), 295
esp_ble_oob_req_reply (C++ function), 251
esp_ble_passkey_reply (C++ function), 250
esp_ble_pcsrk_keys_t (C++ struct), 282
esp_ble_pcsrk_keys_t::counter (C++ member), 283
esp_ble_pcsrk_keys_t::csrk (C++ member), 283
esp_ble_pcsrk_keys_t::sec_level (C++ member), 283
esp_ble_penc_keys_t (C++ struct), 282
esp_ble_penc_keys_t::ediv (C++ member), 282

esp_ble_penc_keys_t::key_size (C++ member), 282
esp_ble_penc_keys_t::ltk (C++ member), 282
esp_ble_penc_keys_t::rand (C++ member), 282
esp_ble_penc_keys_t::sec_level (C++ member), 282
esp_ble_pid_keys_t (C++ struct), 283
esp_ble_pid_keys_t::addr_type (C++ member), 283
esp_ble_pid_keys_t::irk (C++ member), 283
esp_ble_pid_keys_t::static_addr (C++ member), 283
esp_ble_pkt_data_length_params_t (C++ struct), 282
esp_ble_pkt_data_length_params_t::rx_len (C++ member), 282
esp_ble_pkt_data_length_params_t::tx_len (C++ member), 282
esp_ble_power_type_t (C++ enum), 529
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_ADV (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL0 (C++ enumerator), 529
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL1 (C++ enumerator), 529
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL2 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL3 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL4 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL5 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL6 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL7 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_CONN_HDL8 (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_DEFAULT (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_NUM (C++ enumerator), 530
esp_ble_power_type_t::ESP_BLE_PWR_TYPE_SCAN (C++ enumerator), 530
ESP_BLE_PRIM_ADV_INT_MAX (C macro), 237
ESP_BLE_PRIM_ADV_INT_MIN (C macro), 237
esp_ble_remove_bond_device (C++ function), 250
esp_ble_resolve_adv_data (C++ function), 248
esp_ble_sc_oob_req_reply (C++ function), 251
esp_ble_scan_dupilcate_list_flush (C++ function), 526
esp_ble_scan_duplicate_t (C++ enum), 313
esp_ble_scan_duplicate_t::BLE_SCAN_DUPLICATE_DISA (C++ enumerator), 313
esp_ble_scan_duplicate_t::BLE_SCAN_DUPLICATE_ENAB (C++ enumerator), 313
esp_ble_scan_duplicate_t::BLE_SCAN_DUPLICATE_ENAB (C++ enumerator), 313
esp_ble_scan_duplicate_t::BLE_SCAN_DUPLICATE_MAX (C++ enumerator), 314
esp_ble_scan_filter_t (C++ enum), 313
esp_ble_scan_filter_t::BLE_SCAN_FILTER_ALLOW_ALL (C++ enumerator), 313
esp_ble_scan_filter_t::BLE_SCAN_FILTER_ALLOW_ONLY (C++ enumerator), 313
esp_ble_scan_filter_t::BLE_SCAN_FILTER_ALLOW_UND_ (C++ enumerator), 313
esp_ble_scan_filter_t::BLE_SCAN_FILTER_ALLOW_WLIS (C++ enumerator), 313
esp_ble_scan_params_t (C++ struct), 280
esp_ble_scan_params_t::own_addr_type (C++ member), 281
esp_ble_scan_params_t::scan_duplicate (C++ member), 281
esp_ble_scan_params_t::scan_filter_policy (C++ member), 281
esp_ble_scan_params_t::scan_interval (C++ member), 281
esp_ble_scan_params_t::scan_type (C++ member), 281
esp_ble_scan_params_t::scan_window (C++ member), 281
ESP_BLE_SCAN_RSP_DATA_LEN_MAX (C macro), 299
esp_ble_scan_type_t (C++ enum), 313
esp_ble_scan_type_t::BLE_SCAN_TYPE_ACTIVE (C++ enumerator), 313
esp_ble_scan_type_t::BLE_SCAN_TYPE_PASSIVE (C++ enumerator), 313
esp_ble_sec_act_t (C++ enum), 311
esp_ble_sec_act_t::ESP_BLE_SEC_ENCRYPT (C++ enumerator), 311
esp_ble_sec_act_t::ESP_BLE_SEC_ENCRYPT_MITM (C++ enumerator), 311
esp_ble_sec_act_t::ESP_BLE_SEC_ENCRYPT_NO_MITM (C++ enumerator), 311
esp_ble_sec_key_notif_t (C++ struct), 284
esp_ble_sec_key_notif_t::bd_addr (C++ member), 284
esp_ble_sec_key_notif_t::passkey (C++ member), 284
esp_ble_sec_req_t (C++ struct), 284
esp_ble_sec_req_t::bd_addr (C++ member), 284
esp_ble_sec_t (C++ union), 258
esp_ble_sec_t::auth_cmpl (C++ member), 258
esp_ble_sec_t::ble_id_keys (C++ member), 258
esp_ble_sec_t::ble_key (C++ member), 258

esp_ble_sec_t::ble_req (*C++ member*), 258
esp_ble_sec_t::key_notif (*C++ member*), 258
esp_ble_sec_t::oob_data (*C++ member*), 258
esp_ble_set_encryption (*C++ function*), 250
esp_ble_sm_param_t (*C++ enum*), 311
esp_ble_sm_param_t::ESP_BLE_APP_ENC_KEY_SIZE (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_AUTHEN_REQ_MODE (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_CLEAR_STATIC_PASSKEY (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_IOCAP_MODE (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_MAX_KEY_SIZE (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_MAX_PARAM (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_MIN_KEY_SIZE (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_OOB_SUPPORT (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_PASSKEY (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_SET_INIT_KEY (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_SET_RSP_KEY (*C++ enumerator*), 312
esp_ble_sm_param_t::ESP_BLE_SM_SET_STATIC_PASSKEY (*C++ enumerator*), 312
esp_ble_tx_power_get (*C++ function*), 523
esp_ble_tx_power_set (*C++ function*), 523
esp_ble_vendor_cmd_params_t (*C++ struct*), 278
esp_ble_vendor_cmd_params_t::opcode (*C++ member*), 278
esp_ble_vendor_cmd_params_t::p_param_buf (*C++ member*), 278
esp_ble_vendor_cmd_params_t::param_len (*C++ member*), 278
esp_ble_wl_addr_type_t (*C++ enum*), 243
esp_ble_wl_addr_type_t::BLE_WL_ADDR_TYPE_PUBLIC (*C++ enumerator*), 243
esp_ble_wl_addr_type_t::BLE_WL_ADDR_TYPE_RANDOM (*C++ enumerator*), 243
esp_ble_wl_operation_t (*C++ enum*), 315
esp_ble_wl_operation_t::ESP_BLE_WHITELIST_ADD (*C++ enumerator*), 315
esp_ble_wl_operation_t::ESP_BLE_WHITELIST_CLEAR (*C++ enumerator*), 315
esp_ble_wl_operation_t::ESP_BLE_WHITELIST_REMOVE (*C++ enumerator*), 315
esp_bluedroid_deinit (*C++ function*), 244
esp_bluedroid_disable (*C++ function*), 243
esp_bluedroid_enable (*C++ function*), 243
esp_bluedroid_get_status (*C++ function*), 243
esp_bluedroid_init (*C++ function*), 243
ESP_BLUEDROID_STATUS_CHECK (*C macro*), 237
esp_bluedroid_status_t (*C++ enum*), 244
esp_bluedroid_status_t::ESP_BLUEDROID_STATUS_ENABLED (*C++ enumerator*), 244
esp_bluedroid_status_t::ESP_BLUEDROID_STATUS_INITIALIZED (*C++ enumerator*), 244
esp_bluedroid_status_t::ESP_BLUEDROID_STATUS_UNINITIALIZED (*C++ enumerator*), 244
esp_bluetooth_sdp_hdr_overlay_t (*C++ type*), 520
esp_bluetooth_sdp_mas_record_t (*C++ type*), 520
esp_bluetooth_sdp_mns_record_t (*C++ type*), 520
esp_bluetooth_sdp_ops_record_t (*C++ type*), 521
esp_bluetooth_sdp_pce_record_t (*C++ type*), 520
esp_bluetooth_sdp_pse_record_t (*C++ type*), 520
esp_bluetooth_sdp_record_t (*C++ union*), 516
esp_bluetooth_sdp_record_t::hdr (*C++ member*), 516
esp_bluetooth_sdp_record_t::mas (*C++ member*), 516
esp_bluetooth_sdp_record_t::mns (*C++ member*), 516
esp_bluetooth_sdp_record_t::ops (*C++ member*), 516
esp_bluetooth_sdp_record_t::pce (*C++ member*), 516
esp_bluetooth_sdp_record_t::pse (*C++ member*), 516
esp_bluetooth_sdp_record_t::sap (*C++ member*), 516
esp_bluetooth_sdp_sap_record_t (*C++ type*), 521
esp_bluetooth_sdp_types_t (*C++ enum*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_MAP_MAS (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_MAP_MNS (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_OPP_SERVER (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_PBAP_PCE (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_PBAP_PSE (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_RAW (*C++ enumerator*), 522
esp_bluetooth_sdp_types_t::ESP_SDP_TYPE_SAP_SERVER (*C++ enumerator*), 522
esp_blufi_ap_record_t (*C++ struct*), 374
esp_blufi_ap_record_t::rssi (*C++ member*), 375

esp_blufi_ap_record_t::ssid (*C++ member*), 375
ESP_BLUFI_BD_ADDR_LEN (*C macro*), 375
esp_blufi_bd_addr_t (*C++ type*), 375
esp_blufi_callbacks_t (*C++ struct*), 375
esp_blufi_callbacks_t::checksum_func (*C++ member*), 375
esp_blufi_callbacks_t::decrypt_func (*C++ member*), 375
esp_blufi_callbacks_t::encrypt_func (*C++ member*), 375
esp_blufi_callbacks_t::event_cb (*C++ member*), 375
esp_blufi_callbacks_t::negotiate_data_handler (*C++ member*), 375
esp_blufi_cb_event_t (*C++ enum*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_BLE_CONNECT (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_BLE_DISCONNECT (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_DEAUTHENTICATE_STA (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_DEINIT_FINISH (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_GET_WIFI_LIST (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_GET_WIFI_STATUS (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_INIT_FINISH (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_CA_CERT (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_CLIENT_CERT (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_CUSTOM_DATA (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SERVER_CERT (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_SOFTAP_SSID (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_STA_BSSID (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_STA_PASSWD (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_STA_SSID (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_RECV_USERNAME (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_REPORT_ERROR (*C++ enumerator*), 377
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP (*C++ enumerator*), 376
esp_blufi_cb_event_t::ESP_BLUFI_EVENT_SET_WIFI_OPMODE (*C++ enumerator*), 376
esp_blufi_cb_param_t (*C++ union*), 367
esp_blufi_cb_param_t::blufi_connect_evt_param (*C++ struct*), 368
esp_blufi_cb_param_t::blufi_connect_evt_param::cover (*C++ member*), 369
esp_blufi_cb_param_t::blufi_connect_evt_param::remote_bda (*C++ member*), 369
esp_blufi_cb_param_t::blufi_connect_evt_param::server_if (*C++ member*), 369
esp_blufi_cb_param_t::blufi_deinit_finish_evt_param (*C++ struct*), 369
esp_blufi_cb_param_t::blufi_deinit_finish_evt_param::state (*C++ member*), 369
esp_blufi_cb_param_t::blufi_disconnect_evt_param (*C++ struct*), 369
esp_blufi_cb_param_t::blufi_disconnect_evt_param::reason (*C++ member*), 369
esp_blufi_cb_param_t::blufi_get_error_evt_param (*C++ struct*), 369
esp_blufi_cb_param_t::blufi_get_error_evt_param::error_code (*C++ member*), 369
esp_blufi_cb_param_t::blufi_init_finish_evt_param (*C++ struct*), 369
esp_blufi_cb_param_t::blufi_init_finish_evt_param::state (*C++ member*), 369
esp_blufi_cb_param_t::blufi_recv_ca_evt_param (*C++ struct*), 369
esp_blufi_cb_param_t::blufi_recv_ca_evt_param::cert (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_ca_evt_param::cert_len (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param (*C++ struct*), 370
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param::cert (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param::cert_len (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param (*C++ struct*), 370
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param::pkey (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param::pkey_len (*C++ member*), 370
esp_blufi_cb_param_t::blufi_recv_custom_data_evt_param (*C++ struct*), 370
esp_blufi_cb_param_t::blufi_recv_custom_data_evt_param::

(C++ member), 370
esp_blufi_cb_param_t::blufi_recv_custom_data (C++ member), 370
esp_blufi_cb_param_t::blufi_recv_server_cert (C++ member), 370
esp_blufi_cb_param_t::blufi_recv_server_cert (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_server_cert::cert (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_server_cert::cert_len (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_server_pkey (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_server_pkey::pkey (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_server_pkey::pkey_len (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_softap_auth_mode (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_softap_auth_mode::auth_mode (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_softap_channel (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_softap_channel::channel (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num::max_conn_num (C++ member), 371
esp_blufi_cb_param_t::blufi_recv_softap_passwd (C++ struct), 371
esp_blufi_cb_param_t::blufi_recv_softap_passwd::passwd (C++ member), 372
esp_blufi_cb_param_t::blufi_recv_softap_ssid (C++ struct), 372
esp_blufi_cb_param_t::blufi_recv_softap_ssid::ssid (C++ member), 372
esp_blufi_cb_param_t::blufi_recv_sta_bssid (C++ struct), 372
esp_blufi_cb_param_t::blufi_recv_sta_bssid::bssid (C++ member), 372
esp_blufi_cb_param_t::blufi_recv_sta_passwd (C++ struct), 372
esp_blufi_cb_param_t::blufi_recv_sta_passwd::passwd (C++ member), 372
esp_blufi_cb_param_t::blufi_recv_sta_ssid (C++ struct), 372
esp_blufi_cb_param_t::blufi_recv_sta_ssid::ssid (C++ member), 372
esp_blufi_cb_param_t::blufi_recv_username (C++ struct), 373
esp_blufi_cb_param_t::blufi_recv_username::name (C++ member), 373
esp_blufi_cb_param_t::blufi_recv_username::name_len (C++ member), 373
esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param (C++ struct), 373
esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param::op_mode (C++ member), 373
esp_blufi_cb_param_t::client_cert (C++ member), 368
esp_blufi_cb_param_t::client_pkey (C++ member), 368
esp_blufi_cb_param_t::connect (C++ member), 367
esp_blufi_cb_param_t::custom_data (C++ member), 368
esp_blufi_cb_param_t::deinit_finish (C++ member), 367
esp_blufi_cb_param_t::disconnect (C++ member), 367
esp_blufi_cb_param_t::init_finish (C++ member), 367
esp_blufi_cb_param_t::report_error (C++ member), 368
esp_blufi_cb_param_t::server_pkey (C++ member), 368
esp_blufi_cb_param_t::softap_auth_mode (C++ member), 368
esp_blufi_cb_param_t::softap_channel (C++ member), 368
esp_blufi_cb_param_t::softap_max_conn_num (C++ member), 368
esp_blufi_cb_param_t::softap_passwd (C++ member), 368
esp_blufi_cb_param_t::softap_ssid (C++ member), 368
esp_blufi_cb_param_t::sta_bssid (C++ member), 367
esp_blufi_cb_param_t::sta_passwd (C++ member), 368
esp_blufi_cb_param_t::sta_ssid (C++ member), 367
esp_blufi_cb_param_t::username (C++ member), 368
esp_blufi_cb_param_t::wifi_mode (C++ member), 367
esp_blufi_checksum_func_t (C++ type), 376
esp_blufi_decrypt_func_t (C++ type), 376
esp_blufi_deinit_state_t (C++ enum), 378
esp_blufi_deinit_state_t::ESP_BLUFI_DEINIT_FAILED (C++ enumerator), 378
esp_blufi_deinit_state_t::ESP_BLUFI_DEINIT_OK (C++ enumerator), 378
esp_blufi_encrypt_func_t (C++ type), 376
esp_blufi_extra_info_t (C++ struct), 378

esp_blufi_error_state_t::ESP_BLUFI_CALC_MD5_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_CHECKSUM_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_DATA_FORMAT_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_DECRYPT_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_DH_MALLOC_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_DH_PARAM_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_ENCRYPT_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_INIT_SECURITY_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_MAKE_PUBLIC_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_MSG_STATE_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_READ_PARAM_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_SEQUENCE_ERROR (C++ enumerator), 378
esp_blufi_error_state_t::ESP_BLUFI_WIFI_SCAN_FAIL (C++ enumerator), 378
esp_blufi_event_cb_t (C++ type), 375
esp_blufi_extra_info_t (C++ struct), 373
esp_blufi_extra_info_t::softap_authmode (C++ member), 374
esp_blufi_extra_info_t::softap_authmode_set (C++ member), 374
esp_blufi_extra_info_t::softap_channel (C++ member), 374
esp_blufi_extra_info_t::softap_channel_set (C++ member), 374
esp_blufi_extra_info_t::softap_max_conn_num (C++ member), 374
esp_blufi_extra_info_t::softap_max_conn_num_set (C++ member), 374
esp_blufi_extra_info_t::softap_passwd (C++ member), 374
esp_blufi_extra_info_t::softap_passwd_len (C++ member), 374
esp_blufi_extra_info_t::softap_ssid (C++ member), 373
esp_blufi_extra_info_t::softap_ssid_len (C++ member), 374
esp_blufi_extra_info_t::sta_bssid (C++ member), 373
esp_blufi_extra_info_t::sta_bssid_set (C++ member), 373
esp_blufi_extra_info_t::sta_conn_end_reason (C++ member), 374
esp_blufi_extra_info_t::sta_conn_end_reason_set (C++ member), 374
esp_blufi_extra_info_t::sta_conn_rssi (C++ member), 374
esp_blufi_extra_info_t::sta_conn_rssi_set (C++ member), 374
esp_blufi_extra_info_t::sta_max_conn_retry (C++ member), 374
esp_blufi_extra_info_t::sta_max_conn_retry_set (C++ member), 374
esp_blufi_extra_info_t::sta_passwd (C++ member), 373
esp_blufi_extra_info_t::sta_passwd_len (C++ member), 373
esp_blufi_extra_info_t::sta_ssid (C++ member), 373
esp_blufi_extra_info_t::sta_ssid_len (C++ member), 373
esp_blufi_get_version (C++ function), 367
esp_blufi_init_state_t (C++ enum), 377
esp_blufi_init_state_t::ESP_BLUFI_INIT_FAILED (C++ enumerator), 378
esp_blufi_init_state_t::ESP_BLUFI_INIT_OK (C++ enumerator), 378
esp_blufi_negotiate_data_handler_t (C++ type), 375
esp_blufi_profile_deinit (C++ function), 366
esp_blufi_profile_init (C++ function), 366
esp_blufi_register_callbacks (C++ function), 366
esp_blufi_send_custom_data (C++ function), 367
esp_blufi_send_error_info (C++ function), 367
esp_blufi_send_wifi_conn_report (C++ function), 366
esp_blufi_send_wifi_list (C++ function), 367
esp_blufi_sta_conn_state_t (C++ enum), 377
esp_blufi_sta_conn_state_t::ESP_BLUFI_STA_CONN_FA (C++ enumerator), 377
esp_blufi_sta_conn_state_t::ESP_BLUFI_STA_CONN_SU (C++ enumerator), 377
esp_blufi_sta_conn_state_t::ESP_BLUFI_STA_CONNECT (C++ enumerator), 377
esp_blufi_sta_conn_state_t::ESP_BLUFI_STA_NO_IP (C++ enumerator), 377
esp_bredr_sco_datapath_set (C++ function), 523
esp_bredr_tx_power_get (C++ function), 523
esp_bredr_tx_power_set (C++ function), 523
ESP_BT_ACL_2_DH1_ONLY (C macro), 393
ESP_BT_ACL_2_DH3_ONLY (C macro), 393
ESP_BT_ACL_2_DH5_ONLY (C macro), 393
ESP_BT_ACL_3_DH1_ONLY (C macro), 393
ESP_BT_ACL_3_DH3_ONLY (C macro), 393
ESP_BT_ACL_3_DH5_ONLY (C macro), 393
ESP_BT_ACL_DH1_ONLY (C macro), 393
ESP_BT_ACL_DH3_ONLY (C macro), 393
ESP_BT_ACL_DH5_ONLY (C macro), 393

ESP_BT_ACL_DM1_ONLY (*C macro*), 393
ESP_BT_ACL_DM3_ONLY (*C macro*), 393
ESP_BT_ACL_DM5_ONLY (*C macro*), 393
esp_bt_acl_pkt_type_t (*C++ type*), 395
ESP_BT_ACL_PKT_TYPES_MASK_DH1 (*C macro*), 392
ESP_BT_ACL_PKT_TYPES_MASK_DH3 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_DH5 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_DM1 (*C macro*), 392
ESP_BT_ACL_PKT_TYPES_MASK_DM3 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_DM5 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH1 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH3 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_2_DH5 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH1 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH3 (*C macro*), 393
ESP_BT_ACL_PKT_TYPES_MASK_NO_3_DH5 (*C macro*), 393
ESP_BT_COD_FORMAT_TYPE_1 (*C macro*), 395
ESP_BT_COD_FORMAT_TYPE_BIT_MASK (*C macro*), 395
ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET (*C macro*), 395
ESP_BT_COD_MAJOR_DEV_BIT_MASK (*C macro*), 394
ESP_BT_COD_MAJOR_DEV_BIT_OFFSET (*C macro*), 394
esp_bt_cod_major_dev_t (*C++ enum*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_AV (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_COMPUTER (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_HEALTH (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_IMAGING (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_LAN_NAP (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_MISC (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_PERIPHERAL (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_PHONE (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_TOY (*C++ enumerator*), 398
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED (*C++ enumerator*), 399
esp_bt_cod_major_dev_t::ESP_BT_COD_MAJOR_DEV_WEAR (*C++ enumerator*), 398
ESP_BT_COD_MINOR_DEV_BIT_MASK (*C macro*), 395
ESP_BT_COD_MINOR_DEV_BIT_OFFSET (*C macro*), 395
esp_bt_cod_mode_t (*C++ enum*), 396
esp_bt_cod_mode_t::ESP_BT_CLR_COD_SERVICE_CLASS (*C++ enumerator*), 396
esp_bt_cod_mode_t::ESP_BT_INIT_COD (*C++ enumerator*), 396
esp_bt_cod_mode_t::ESP_BT_SET_COD_ALL (*C++ enumerator*), 396
esp_bt_cod_mode_t::ESP_BT_SET_COD_MAJOR_MINOR (*C++ enumerator*), 396
esp_bt_cod_mode_t::ESP_BT_SET_COD_SERVICE_CLASS (*C++ enumerator*), 396
ESP_BT_COD_SRVC_BIT_MASK (*C macro*), 394
ESP_BT_COD_SRVC_BIT_OFFSET (*C macro*), 394
esp_bt_cod_srvc_t (*C++ enum*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_AUDIO (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_CAPTURING (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_INFORMATION (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_LMTD_DISCOVER (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_NETWORKING (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_NONE (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_OBJ_TRANSFER (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_POSITIONING (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_RENDERING (*C++ enumerator*), 397
esp_bt_cod_srvc_t::ESP_BT_COD_SRVC_TELEPHONY (*C++ enumerator*), 397
esp_bt_cod_t (*C++ struct*), 390
esp_bt_cod_t::major (*C++ member*), 390
esp_bt_cod_t::minor (*C++ member*), 390
esp_bt_cod_t::reserved_2 (*C++ member*), 390
esp_bt_cod_t::reserved_8 (*C++ member*), 390
esp_bt_cod_t::service (*C++ member*), 390
esp_bt_connection_mode_t (*C++ enum*), 396
esp_bt_connection_mode_t::ESP_BT_CONNECTABLE (*C++ enumerator*), 396
esp_bt_connection_mode_t::ESP_BT_NON_CONNECTABLE (*C++ enumerator*), 396
ESP_BT_CONTROLLER_CONFIG_MAGIC_VAL (*C macro*), 528
esp_bt_controller_config_t (*C++ struct*), 526
esp_bt_controller_config_t::auto_latency

(C++ member), 527
esp_bt_controller_config_t::ble_max_conn (C++ member), 527
esp_bt_controller_config_t::ble_sca (C++ member), 527
esp_bt_controller_config_t::ble_scan_backoff (C++ member), 527
esp_bt_controller_config_t::bt_legacy_auth_vs_evt (C++ member), 527
esp_bt_controller_config_t::bt_max_acl_conn (C++ member), 527
esp_bt_controller_config_t::bt_max_sync_conn (C++ member), 527
esp_bt_controller_config_t::bt_sco_datapath (C++ member), 527
esp_bt_controller_config_t::controller_debug_flag (C++ member), 527
esp_bt_controller_config_t::controller_task_prio (C++ member), 526
esp_bt_controller_config_t::controller_task_stack_size (C++ member), 526
esp_bt_controller_config_t::dup_list_refresh_period (C++ member), 527
esp_bt_controller_config_t::hci_uart_baudrate (C++ member), 526
esp_bt_controller_config_t::hci_uart_no (C++ member), 526
esp_bt_controller_config_t::hli (C++ member), 527
esp_bt_controller_config_t::magic (C++ member), 528
esp_bt_controller_config_t::mesh_adv_size (C++ member), 527
esp_bt_controller_config_t::mode (C++ member), 527
esp_bt_controller_config_t::normal_adv_size (C++ member), 526
esp_bt_controller_config_t::pcm_polar (C++ member), 527
esp_bt_controller_config_t::pcm_role (C++ member), 527
esp_bt_controller_config_t::scan_duplicate_mode (C++ member), 526
esp_bt_controller_config_t::scan_duplicate_type (C++ member), 526
esp_bt_controller_config_t::send_adv_reserved_size (C++ member), 527
esp_bt_controller_deinit (C++ function), 523
esp_bt_controller_disable (C++ function), 524
esp_bt_controller_enable (C++ function), 523
esp_bt_controller_get_status (C++ function), 524
esp_bt_controller_init (C++ function), 523
esp_bt_controller_mem_release (C++ function), 524
esp_bt_controller_status_t (C++ enum), 529
esp_bt_controller_status_t::ESP_BT_CONTROLLER_STA (C++ enumerator), 529
esp_bt_controller_status_t::ESP_BT_CONTROLLER_STA (C++ enumerator), 529
esp_bt_controller_status_t::ESP_BT_CONTROLLER_STA (C++ enumerator), 529
esp_bt_controller_status_t::ESP_BT_CONTROLLER_STA (C++ enumerator), 529
esp_bt_dev_get_address (C++ function), 244
esp_bt_dev_set_device_name (C++ function), 244
esp_bt_dev_type_t (C++ enum), 242
esp_bt_dev_type_t::ESP_BT_DEVICE_TYPE_BLE (C++ enumerator), 242
esp_bt_dev_type_t::ESP_BT_DEVICE_TYPE_BREDR (C++ enumerator), 242
esp_bt_dev_type_t::ESP_BT_DEVICE_TYPE_DUMO (C++ enumerator), 242
esp_bt_discovery_mode_t (C++ enum), 396
esp_bt_discovery_mode_t::ESP_BT_GENERAL_DISCOVERA (C++ enumerator), 396
esp_bt_discovery_mode_t::ESP_BT_LIMITED_DISCOVERA (C++ enumerator), 396
esp_bt_discovery_mode_t::ESP_BT_NON_DISCOVERABLE (C++ enumerator), 396
esp_bt_duplicate_exceptional_subcode_type_t (C++ enum), 315
esp_bt_duplicate_exceptional_subcode_type_t::ESP_ (C++ enumerator), 315
esp_bt_duplicate_exceptional_subcode_type_t::ESP_ (C++ enumerator), 315
esp_bt_duplicate_exceptional_subcode_type_t::ESP_ (C++ enumerator), 315
esp_bt_eir_data_t (C++ struct), 391
esp_bt_eir_data_t::fec_required (C++ member), 391
esp_bt_eir_data_t::flag (C++ member), 391
esp_bt_eir_data_t::include_name (C++ member), 391
esp_bt_eir_data_t::include_txpower (C++ member), 391
esp_bt_eir_data_t::include_uuid (C++ member), 391
esp_bt_eir_data_t::manufacturer_len (C++ member), 391
esp_bt_eir_data_t::p_manufacturer_data (C++ member), 391
esp_bt_eir_data_t::p_url (C++ member), 391
esp_bt_eir_data_t::url_len (C++ member), 391
ESP_BT_EIR_FLAG_BREDR_NOT_SPT (C macro), 394
ESP_BT_EIR_FLAG_DMT_CONTROLLER_SPT (C macro), 394
ESP_BT_EIR_FLAG_DMT_HOST_SPT (C macro),

394
ESP_BT_EIR_FLAG_GEN_DISC (*C macro*), 394
ESP_BT_EIR_FLAG_LIMIT_DISC (*C macro*), 393
ESP_BT_EIR_MAX_LEN (*C macro*), 394
ESP_BT_EIR_TYPE_CMPL_128BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_CMPL_16BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_CMPL_32BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_CMPL_LOCAL_NAME (*C macro*), 392
ESP_BT_EIR_TYPE_FLAGS (*C macro*), 392
ESP_BT_EIR_TYPE_INCMPL_128BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_INCMPL_16BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_INCMPL_32BITS_UUID (*C macro*), 392
ESP_BT_EIR_TYPE_MANU_SPECIFIC (*C macro*), 392
ESP_BT_EIR_TYPE_MAX_NUM (*C macro*), 392
ESP_BT_EIR_TYPE_SHORT_LOCAL_NAME (*C macro*), 392
esp_bt_eir_type_t (*C++ type*), 395
ESP_BT_EIR_TYPE_TX_POWER_LEVEL (*C macro*), 392
ESP_BT_EIR_TYPE_URL (*C macro*), 392
esp_bt_gap_afh_channels (*C++ type*), 395
ESP_BT_GAP_AFH_CHANNELS_LEN (*C macro*), 391
esp_bt_gap_cancel_discovery (*C++ function*), 380
esp_bt_gap_cb_event_t (*C++ enum*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_ACL_CONN_CMPL_STAT_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_ACL_DISCONN_CMPL_STAT_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_ACL_PKT_TYPE_CHANGED_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_AUTH_CMPL_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_CFM_REQ_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_CONFIG_EIR_DATA_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_DISC_RES_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_DISC_STATE_CHANGED_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_EVT_MAX (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_KEY_NOTIF_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_KEY_REQ_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_MODE_CHG_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_PIN_REQ_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_QOS_CMPL_EVT (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_READ_REMOTE_NAM (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_READ_RSSI_DELTA (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_REMOVE_BOND_DEV (*C++ enumerator*), 400
esp_bt_gap_cb_event_t::ESP_BT_GAP_RMT_SRVC_REC_EV (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_RMT_SRVCS_EVT (*C++ enumerator*), 399
esp_bt_gap_cb_event_t::ESP_BT_GAP_SET_AFH_CHANNEL (*C++ enumerator*), 400
esp_bt_gap_cb_param_t (*C++ union*), 383
esp_bt_gap_cb_param_t::acl_conn_cmpl_stat (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_conn_cmpl_stat_param (*C++ struct*), 385
esp_bt_gap_cb_param_t::acl_conn_cmpl_stat_param:: (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_conn_cmpl_stat_param:: (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_conn_cmpl_stat_param:: (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_disconn_cmpl_stat (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_disconn_cmpl_stat_para (*C++ struct*), 385
esp_bt_gap_cb_param_t::acl_disconn_cmpl_stat_para (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_disconn_cmpl_stat_para (*C++ member*), 385
esp_bt_gap_cb_param_t::acl_disconn_cmpl_stat_para (*C++ member*), 385
esp_bt_gap_cb_param_t::auth_cmpl (*C++ member*), 384
esp_bt_gap_cb_param_t::auth_cmpl_param (*C++ struct*), 385
esp_bt_gap_cb_param_t::auth_cmpl_param::bda (*C++ member*), 385
esp_bt_gap_cb_param_t::auth_cmpl_param::device_na (*C++ member*), 386
esp_bt_gap_cb_param_t::auth_cmpl_param::stat (*C++ member*), 385
esp_bt_gap_cb_param_t::bt_remove_bond_dev_cmpl_ev (*C++ struct*), 386
esp_bt_gap_cb_param_t::bt_remove_bond_dev_cmpl_ev (*C++ member*), 386
esp_bt_gap_cb_param_t::bt_remove_bond_dev_cmpl_ev (*C++ member*), 386
esp_bt_gap_cb_param_t::cfm_req (*C++ member*), 384
esp_bt_gap_cb_param_t::cfm_req_param (*C++ struct*), 386
esp_bt_gap_cb_param_t::cfm_req_param::bda

(C++ member), 386
esp_bt_gap_cb_param_t::cfm_req_param::num_val (C++ member), 386
esp_bt_gap_cb_param_t::config_eir_data (C++ member), 384
esp_bt_gap_cb_param_t::config_eir_data_param (C++ struct), 386
esp_bt_gap_cb_param_t::config_eir_data_param::eir_data (C++ member), 386
esp_bt_gap_cb_param_t::config_eir_data_param::eir_data_len (C++ member), 386
esp_bt_gap_cb_param_t::config_eir_data_param::stat (C++ member), 386
esp_bt_gap_cb_param_t::disc_res (C++ member), 383
esp_bt_gap_cb_param_t::disc_res_param (C++ struct), 386
esp_bt_gap_cb_param_t::disc_res_param::bda (C++ member), 387
esp_bt_gap_cb_param_t::disc_res_param::num_prop (C++ member), 387
esp_bt_gap_cb_param_t::disc_res_param::prop (C++ member), 387
esp_bt_gap_cb_param_t::disc_st_chg (C++ member), 384
esp_bt_gap_cb_param_t::disc_state_changed_param (C++ struct), 387
esp_bt_gap_cb_param_t::disc_state_changed_param::state (C++ member), 387
esp_bt_gap_cb_param_t::key_notif (C++ member), 384
esp_bt_gap_cb_param_t::key_notif_param (C++ struct), 387
esp_bt_gap_cb_param_t::key_notif_param::bda (C++ member), 387
esp_bt_gap_cb_param_t::key_notif_param::passkey (C++ member), 387
esp_bt_gap_cb_param_t::key_req (C++ member), 384
esp_bt_gap_cb_param_t::key_req_param (C++ struct), 387
esp_bt_gap_cb_param_t::key_req_param::bda (C++ member), 387
esp_bt_gap_cb_param_t::mode_chg (C++ member), 384
esp_bt_gap_cb_param_t::mode_chg_param (C++ struct), 387
esp_bt_gap_cb_param_t::mode_chg_param::bda (C++ member), 387
esp_bt_gap_cb_param_t::mode_chg_param::mode (C++ member), 387
esp_bt_gap_cb_param_t::pin_req (C++ member), 384
esp_bt_gap_cb_param_t::pin_req_param (C++ struct), 388
esp_bt_gap_cb_param_t::pin_req_param::bda (C++ member), 388
esp_bt_gap_cb_param_t::pin_req_param::min_16_digit (C++ member), 388
esp_bt_gap_cb_param_t::qos_cmpl (C++ member), 384
esp_bt_gap_cb_param_t::qos_cmpl_param (C++ struct), 388
esp_bt_gap_cb_param_t::qos_cmpl_param::bda (C++ member), 388
esp_bt_gap_cb_param_t::qos_cmpl_param::stat (C++ member), 388
esp_bt_gap_cb_param_t::qos_cmpl_param::t_poll (C++ member), 388
esp_bt_gap_cb_param_t::read_rmt_name (C++ member), 384
esp_bt_gap_cb_param_t::read_rmt_name_param (C++ struct), 388
esp_bt_gap_cb_param_t::read_rmt_name_param::bda (C++ member), 388
esp_bt_gap_cb_param_t::read_rmt_name_param::rmt_name (C++ member), 388
esp_bt_gap_cb_param_t::read_rmt_name_param::stat (C++ member), 388
esp_bt_gap_cb_param_t::read_rssi_delta (C++ member), 384
esp_bt_gap_cb_param_t::read_rssi_delta_param (C++ struct), 388
esp_bt_gap_cb_param_t::read_rssi_delta_param::bda (C++ member), 389
esp_bt_gap_cb_param_t::read_rssi_delta_param::rssi_delta (C++ member), 389
esp_bt_gap_cb_param_t::read_rssi_delta_param::stat (C++ member), 389
esp_bt_gap_cb_param_t::remove_bond_dev_cmpl (C++ member), 384
esp_bt_gap_cb_param_t::rmt_srvc_rec (C++ member), 384
esp_bt_gap_cb_param_t::rmt_srvc_rec_param (C++ struct), 389
esp_bt_gap_cb_param_t::rmt_srvc_rec_param::bda (C++ member), 389
esp_bt_gap_cb_param_t::rmt_srvc_rec_param::stat (C++ member), 389
esp_bt_gap_cb_param_t::rmt_srvcs (C++ member), 384
esp_bt_gap_cb_param_t::rmt_srvcs_param (C++ struct), 389
esp_bt_gap_cb_param_t::rmt_srvcs_param::bda (C++ member), 389
esp_bt_gap_cb_param_t::rmt_srvcs_param::num_uuids (C++ member), 389
esp_bt_gap_cb_param_t::rmt_srvcs_param::stat (C++ member), 389
esp_bt_gap_cb_param_t::rmt_srvcs_param::uuid_list (C++ member), 389
esp_bt_gap_cb_param_t::set_acl_pkt_types (C++ member), 384
esp_bt_gap_cb_param_t::set_acl_pkt_types_param (C++ struct), 389
esp_bt_gap_cb_param_t::set_acl_pkt_types_param::b

(C++ *member*), 390
esp_bt_gap_cb_param_t::set_acl_pkt_types (C++ *member*), 390
esp_bt_gap_cb_param_t::set_acl_pkt_types (C++ *member*), 389
esp_bt_gap_cb_param_t::set_afh_channels (C++ *member*), 384
esp_bt_gap_cb_param_t::set_afh_channels_stat (C++ *struct*), 390
esp_bt_gap_cb_param_t::set_afh_channels_stat::stat (C++ *member*), 390
esp_bt_gap_cb_t (C++ *type*), 396
esp_bt_gap_config_eir_data (C++ *function*), 381
esp_bt_gap_dev_prop_t (C++ *struct*), 390
esp_bt_gap_dev_prop_t::len (C++ *member*), 390
esp_bt_gap_dev_prop_t::type (C++ *member*), 390
esp_bt_gap_dev_prop_t::val (C++ *member*), 391
esp_bt_gap_dev_prop_type_t (C++ *enum*), 396
esp_bt_gap_dev_prop_type_t::ESP_BT_GAP_DEV_PROP_BDNAME (C++ *enumerator*), 397
esp_bt_gap_dev_prop_type_t::ESP_BT_GAP_DEV_PROP_COD (C++ *enumerator*), 397
esp_bt_gap_dev_prop_type_t::ESP_BT_GAP_DEV_PROP_EIR (C++ *enumerator*), 397
esp_bt_gap_dev_prop_type_t::ESP_BT_GAP_DEV_PROP_RSSI (C++ *enumerator*), 397
esp_bt_gap_discovery_state_t (C++ *enum*), 399
esp_bt_gap_discovery_state_t::ESP_BT_GAP_DISCOVERY_STARTED (C++ *enumerator*), 399
esp_bt_gap_discovery_state_t::ESP_BT_GAP_DISCOVERY_STOPPED (C++ *enumerator*), 399
ESP_BT_GAP_EIR_DATA_LEN (C *macro*), 392
esp_bt_gap_get_bond_device_list (C++ *function*), 382
esp_bt_gap_get_bond_device_num (C++ *function*), 381
esp_bt_gap_get_cod (C++ *function*), 381
esp_bt_gap_get_cod_format_type (C++ *function*), 379
esp_bt_gap_get_cod_major_dev (C++ *function*), 379
esp_bt_gap_get_cod_minor_dev (C++ *function*), 379
esp_bt_gap_get_cod_srvc (C++ *function*), 379
esp_bt_gap_get_remote_service_record (C++ *function*), 380
esp_bt_gap_get_remote_services (C++ *function*), 380
esp_bt_gap_is_valid_cod (C++ *function*), 379
ESP_BT_GAP_MAX_BDNAME_LEN (C *macro*), 391
ESP_BT_GAP_MAX_INQ_LEN (C *macro*), 395
ESP_BT_GAP_MIN_INQ_LEN (C *macro*), 395
esp_bt_gap_pin_reply (C++ *function*), 382
esp_bt_gap_pkt_type_t (C++ *type*), 396
esp_bt_gap_read_remote_name (C++ *function*), 383
esp_bt_gap_read_rssi_delta (C++ *function*), 381
esp_bt_gap_register_callback (C++ *function*), 379
esp_bt_gap_remove_bond_device (C++ *function*), 381
esp_bt_gap_resolve_eir_data (C++ *function*), 380
ESP_BT_GAP_RSSI_HIGH_THRLD (C *macro*), 391
ESP_BT_GAP_RSSI_LOW_THRLD (C *macro*), 391
esp_bt_gap_set_acl_pkt_types (C++ *function*), 383
esp_bt_gap_set_afh_channels (C++ *function*), 383
esp_bt_gap_set_cod (C++ *function*), 381
esp_bt_gap_set_pin (C++ *function*), 382
esp_bt_gap_set_qos (C++ *function*), 383
esp_bt_gap_set_scan_mode (C++ *function*), 379
esp_bt_gap_set_security_param (C++ *function*), 382
esp_bt_gap_ssp_confirm_reply (C++ *function*), 383
esp_bt_gap_ssp_passkey_reply (C++ *function*), 382
esp_bt_gap_start_discovery (C++ *function*), 380
ESP_BT_GAP_TPOLL_DFT (C *macro*), 395
ESP_BT_GAP_TPOLL_MAX (C *macro*), 395
ESP_BT_GAP_TPOLL_MIN (C *macro*), 395
ESP_BT_HF_CLIENT_AUDIO_EXAPL_LEN (C *macro*), 463
ESP_BT_HF_CLIENT_NUMBER_LEN (C *macro*), 463
ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN (C *macro*), 463
esp_bt_hid_device_connect (C++ *function*), 483
esp_bt_hid_device_deinit (C++ *function*), 482
esp_bt_hid_device_disconnect (C++ *function*), 483
esp_bt_hid_device_init (C++ *function*), 482
esp_bt_hid_device_register_app (C++ *function*), 483
esp_bt_hid_device_register_callback (C++ *function*), 482
esp_bt_hid_device_report_error (C++ *function*), 484
esp_bt_hid_device_send_report (C++ *function*), 483
esp_bt_hid_device_unregister_app (C++ *function*), 483
esp_bt_hid_device_virtual_cable_unplug (C++ *function*), 484
esp_bt_hid_host_connect (C++ *function*), 495

esp_bt_hid_host_deinit (*C++ function*), 494
esp_bt_hid_host_disconnect (*C++ function*), 495
esp_bt_hid_host_get_idle (*C++ function*), 495
esp_bt_hid_host_get_protocol (*C++ function*), 495
esp_bt_hid_host_get_report (*C++ function*), 496
esp_bt_hid_host_init (*C++ function*), 494
esp_bt_hid_host_register_callback (*C++ function*), 494
esp_bt_hid_host_send_data (*C++ function*), 496
esp_bt_hid_host_set_idle (*C++ function*), 496
esp_bt_hid_host_set_info (*C++ function*), 495
esp_bt_hid_host_set_protocol (*C++ function*), 495
esp_bt_hid_host_set_report (*C++ function*), 496
esp_bt_hid_host_virtual_cable_unplug (*C++ function*), 495
esp_bt_inq_mode_t (*C++ enum*), 400
esp_bt_inq_mode_t::ESP_BT_INQ_MODE_GENERAL_INQUIRY (*C++ enumerator*), 400
esp_bt_inq_mode_t::ESP_BT_INQ_MODE_LIMITED_INQUIRY (*C++ enumerator*), 400
ESP_BT_IO_CAP_IN (*C macro*), 394
ESP_BT_IO_CAP_IO (*C macro*), 394
ESP_BT_IO_CAP_NONE (*C macro*), 394
ESP_BT_IO_CAP_OUT (*C macro*), 394
esp_bt_io_cap_t (*C++ type*), 395
esp_bt_l2cap_cb_event_t (*C++ enum*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_CL_INIT_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_CLOSE_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_INIT_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_OPEN_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_SRV_STOP_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_START_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_event_t::ESP_BT_L2CAP_UNINIT_EVT (*C++ enumerator*), 514
esp_bt_l2cap_cb_param_t (*C++ union*), 510
esp_bt_l2cap_cb_param_t::cl_init (*C++ member*), 511
esp_bt_l2cap_cb_param_t::close (*C++ member*), 510
esp_bt_l2cap_cb_param_t::init (*C++ member*), 510
esp_bt_l2cap_cb_param_t::l2cap_cl_init_evt_param (*C++ struct*), 511
esp_bt_l2cap_cb_param_t::l2cap_cl_init_evt_param: (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_cl_init_evt_param: (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_cl_init_evt_param: (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_close_evt_param (*C++ struct*), 511
esp_bt_l2cap_cb_param_t::l2cap_close_evt_param::a (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_close_evt_param::h (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_close_evt_param::s (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_init_evt_param (*C++ struct*), 511
esp_bt_l2cap_cb_param_t::l2cap_init_evt_param::st (*C++ member*), 511
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param (*C++ struct*), 512
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param::fd (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param::ha (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param::re (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param::st (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_open_evt_param::tx (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_srv_stop_evt_param (*C++ struct*), 512
esp_bt_l2cap_cb_param_t::l2cap_srv_stop_evt_param (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_srv_stop_evt_param (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_start_evt_param (*C++ struct*), 512
esp_bt_l2cap_cb_param_t::l2cap_start_evt_param::h (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_start_evt_param::s (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_start_evt_param::s (*C++ member*), 512
esp_bt_l2cap_cb_param_t::l2cap_uninit_evt_param (*C++ struct*), 513
esp_bt_l2cap_cb_param_t::l2cap_uninit_evt_param:: (*C++ member*), 513
esp_bt_l2cap_cb_param_t::open (*C++ member*), 510
esp_bt_l2cap_cb_param_t::srv_stop (*C++ member*), 511
esp_bt_l2cap_cb_param_t::start (*C++ member*), 511
esp_bt_l2cap_cb_param_t::uninit (*C++ member*), 510
esp_bt_l2cap_cb_t (*C++ type*), 513
esp_bt_l2cap_cntl_flags_t (*C++ type*), 513

esp_bt_l2cap_connect (C++ function), 509
esp_bt_l2cap_deinit (C++ function), 509
esp_bt_l2cap_init (C++ function), 509
esp_bt_l2cap_register_callback (C++ function), 509
ESP_BT_L2CAP_SEC_AUTHENTICATE (C macro), 513
ESP_BT_L2CAP_SEC_AUTHORIZE (C macro), 513
ESP_BT_L2CAP_SEC_ENCRYPT (C macro), 513
ESP_BT_L2CAP_SEC_NONE (C macro), 513
esp_bt_l2cap_start_srv (C++ function), 509
esp_bt_l2cap_status_t (C++ enum), 513
esp_bt_l2cap_status_t::ESP_BT_L2CAP_BUSY (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_FAILURE (C++ enumerator), 513
esp_bt_l2cap_status_t::ESP_BT_L2CAP_NEED_DEINIT (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_NEED_INIT (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_NO_CONNECTION (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_NO_RESOURCE (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_NO_SERVER (C++ enumerator), 514
esp_bt_l2cap_status_t::ESP_BT_L2CAP_SUCCESS (C++ enumerator), 513
esp_bt_l2cap_stop_all_srv (C++ function), 509
esp_bt_l2cap_stop_srv (C++ function), 510
esp_bt_l2cap_vfs_register (C++ function), 510
esp_bt_l2cap_vfs_unregister (C++ function), 510
esp_bt_mem_release (C++ function), 525
esp_bt_mode_t (C++ enum), 528
esp_bt_mode_t::ESP_BT_MODE_BLE (C++ enumerator), 528
esp_bt_mode_t::ESP_BT_MODE_BTDM (C++ enumerator), 528
esp_bt_mode_t::ESP_BT_MODE_CLASSIC_BT (C++ enumerator), 528
esp_bt_mode_t::ESP_BT_MODE_IDLE (C++ enumerator), 528
ESP_BT_OCTET16_LEN (C macro), 237
esp_bt_octet16_t (C++ type), 238
ESP_BT_OCTET8_LEN (C macro), 237
esp_bt_octet8_t (C++ type), 238
ESP_BT_PIN_CODE_LEN (C macro), 394
esp_bt_pin_code_t (C++ type), 395
esp_bt_pin_type_t (C++ enum), 398
esp_bt_pin_type_t::ESP_BT_PIN_TYPE_FIXED (C++ enumerator), 398
esp_bt_pin_type_t::ESP_BT_PIN_TYPE_VARIABLE (C++ enumerator), 398
ESP_BT_PM_MD_ACTIVE (C macro), 394
ESP_BT_PM_MD_HOLD (C macro), 394
ESP_BT_PM_MD_PARK (C macro), 394
ESP_BT_PM_MD_SNIFF (C macro), 394
esp_bt_pm_mode_t (C++ type), 395
esp_bt_sleep_disable (C++ function), 525
esp_bt_sleep_enable (C++ function), 525
esp_bt_sp_param_t (C++ enum), 398
esp_bt_sp_param_t::ESP_BT_SP_IOCAP_MODE (C++ enumerator), 398
ESP_BT_STATUS_BASE_FOR_HCI_ERR (C macro), 237
esp_bt_status_t (C++ enum), 238
esp_bt_status_t::ESP_BT_STATUS_AUTH_FAILURE (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_AUTH_REJECTED (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_BUSY (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_CONTROL_LE_DATA_LE (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_DONE (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_EIR_TOO_LARGE (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_ERR_ILLEGAL_PARAME (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_FAIL (C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_HCI_AUTH_FAILURE (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_CHAN_CLASSIF_N (C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_COMMAND_DISALL (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_CONN_CAUSE_LOC (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_CONN_FAILED_ES (C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_CONN_TOUT_DUE_ (C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_CONNECTION_EXI (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_CONNECTION_TOU (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_CONTROLLER_BUS (C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_DIFF_TRANSACTI (C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_DIRECTED_ADVER (C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_ENCRY_MODE_NOT (C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_HOST_BUSY_PAIR (C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_HOST_REJECT_DE (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_HOST_REJECT_RE (C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_HOST_REJECT_SE

(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_HOST_TIMEOUT
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_HW_FAILURE
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_ILLEGAL_COMMAND
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_HCI_ILLEGAL_PARAMETER_FMT
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_INQ_RSP_DATA_TOO_LARGE
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_INSTANT_PASSED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_INSUFFICIENT_SECURITY
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_INVALID_LMP_PARAM
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_KEY_MISSING
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_LMP_ERR_TRANS_COLLISION
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_LMP_PDU_NOT_ALLOWED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_LMP_RESPONSE_TIMEOUT
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_MAC_CONNECTION_FAILED
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_MAX_NUM_OF_CONNECTIONS
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_MAX_NUM_OF_SCOS
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_MEMORY_FULL
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_NO_CONNECTION
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_PAGE_TIMEOUT
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_PAIRING_NOT_ALLOWED
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_PAIRING_WITH_UNIT_KEY_NOT_SUPPORTED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_PARAM_OUT_OF_RANGE
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_PEER_LOW_RESOURCES
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_PEER_POWER_OFF
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_PEER_USER
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_QOS_NOT_SUPPORTED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_QOS_REJECTED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_QOS_UNACCEPTABLE_PARAM
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_REJ_NO_SUITABLE_CHANNEL
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_REPEATED_ATTEMPTS
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_RESERVED_SLOT_
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_ROLE_CHANGE_NO
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_ROLE_SWITCH_FA
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_ROLE_SWITCH_PE
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_SCO_AIR_MODE
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_SCO_INTERVAL_R
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_SCO_OFFSET_REJ
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_SIMPLE_PAIRING
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_SUCCESS
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_HCI_UNACCEPT_CONN_
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_UNDEFINED_0x2B
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNDEFINED_0x31
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_UNDEFINED_0x33
(C++ enumerator), 242
esp_bt_status_t::ESP_BT_STATUS_HCI_UNIT_KEY_USED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNKNOWN_LMP_PD
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNSPECIFIED
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNSUPPORTED_LM
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNSUPPORTED_RE
(C++ enumerator), 241
esp_bt_status_t::ESP_BT_STATUS_HCI_UNSUPPORTED_VA
(C++ enumerator), 240
esp_bt_status_t::ESP_BT_STATUS_HCI_INVALID_STATIC_RAN
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_MEMORY_FULL
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_NOMEM
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_NOT_READY
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_PARAM_OUT_OF_RANGE
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_PARM_INVALID
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_PEER_LE_DATA_LEN_U
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_PENDING
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_RMT_DEV_DOWN
(C++ enumerator), 239
esp_bt_status_t::ESP_BT_STATUS_SUCCESS

(C++ *enumerator*), 238
esp_bt_status_t::ESP_BT_STATUS_TIMEOUT (C++ *enumerator*), 239
esp_bt_status_t::ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL (C++ *enumerator*), 239
esp_bt_status_t::ESP_BT_STATUS_UNHANDLED (C++ *enumerator*), 239
esp_bt_status_t::ESP_BT_STATUS_UNSUPPORTED (C++ *enumerator*), 239
esp_bt_uuid_t (C++ *struct*), 236
esp_bt_uuid_t::len (C++ *member*), 236
esp_bt_uuid_t::uuid (C++ *member*), 237
esp_bt_uuid_t::uuid128 (C++ *member*), 237
esp_bt_uuid_t::uuid16 (C++ *member*), 236
esp_bt_uuid_t::uuid32 (C++ *member*), 237
esp_chip_id_t (C++ *enum*), 1829
esp_chip_id_t::ESP_CHIP_ID_ESP32 (C++ *enumerator*), 1829
esp_chip_id_t::ESP_CHIP_ID_ESP32C2 (C++ *enumerator*), 1830
esp_chip_id_t::ESP_CHIP_ID_ESP32C3 (C++ *enumerator*), 1830
esp_chip_id_t::ESP_CHIP_ID_ESP32S2 (C++ *enumerator*), 1829
esp_chip_id_t::ESP_CHIP_ID_ESP32S3 (C++ *enumerator*), 1830
esp_chip_id_t::ESP_CHIP_ID_INVALID (C++ *enumerator*), 1830
esp_chip_info (C++ *function*), 2088
esp_chip_info_t (C++ *struct*), 2088
esp_chip_info_t::cores (C++ *member*), 2089
esp_chip_info_t::features (C++ *member*), 2088
esp_chip_info_t::model (C++ *member*), 2088
esp_chip_info_t::revision (C++ *member*), 2089
esp_chip_model_t (C++ *enum*), 2089
esp_chip_model_t::CHIP_ESP32 (C++ *enumerator*), 2089
esp_chip_model_t::CHIP_ESP32C2 (C++ *enumerator*), 2089
esp_chip_model_t::CHIP_ESP32C3 (C++ *enumerator*), 2089
esp_chip_model_t::CHIP_ESP32H2 (C++ *enumerator*), 2089
esp_chip_model_t::CHIP_ESP32S2 (C++ *enumerator*), 2089
esp_chip_model_t::CHIP_ESP32S3 (C++ *enumerator*), 2089
esp_console_cmd_func_t (C++ *type*), 1849
esp_console_cmd_register (C++ *function*), 1844
esp_console_cmd_t (C++ *struct*), 1848
esp_console_cmd_t::argtable (C++ *member*), 1848
esp_console_cmd_t::command (C++ *member*), 1848
esp_console_cmd_t::func (C++ *member*), 1848
esp_console_cmd_t::help (C++ *member*), 1848
esp_console_cmd_t::hint (C++ *member*), 1848
ESP_CONSOLE_CONFIG_DEFAULT (C *macro*), 1848
esp_console_config_t (C++ *struct*), 1846
esp_console_config_t::hint_bold (C++ *member*), 1847
esp_console_config_t::hint_color (C++ *member*), 1847
esp_console_config_t::max_cmdline_args (C++ *member*), 1847
esp_console_config_t::max_cmdline_length (C++ *member*), 1847
esp_console_deinit (C++ *function*), 1844
ESP_CONSOLE_DEV_UART_CONFIG_DEFAULT (C *macro*), 1848
esp_console_dev_uart_config_t (C++ *struct*), 1847
esp_console_dev_uart_config_t::baud_rate (C++ *member*), 1847
esp_console_dev_uart_config_t::channel (C++ *member*), 1847
esp_console_dev_uart_config_t::rx_gpio_num (C++ *member*), 1848
esp_console_dev_uart_config_t::tx_gpio_num (C++ *member*), 1847
esp_console_get_completion (C++ *function*), 1845
esp_console_get_hint (C++ *function*), 1845
esp_console_init (C++ *function*), 1844
esp_console_new_repl_uart (C++ *function*), 1846
esp_console_register_help_command (C++ *function*), 1846
ESP_CONSOLE_REPL_CONFIG_DEFAULT (C *macro*), 1848
esp_console_repl_config_t (C++ *struct*), 1847
esp_console_repl_config_t::history_save_path (C++ *member*), 1847
esp_console_repl_config_t::max_cmdline_length (C++ *member*), 1847
esp_console_repl_config_t::max_history_len (C++ *member*), 1847
esp_console_repl_config_t::prompt (C++ *member*), 1847
esp_console_repl_config_t::task_priority (C++ *member*), 1847
esp_console_repl_config_t::task_stack_size (C++ *member*), 1847
esp_console_repl_s (C++ *struct*), 1848
esp_console_repl_s::del (C++ *member*), 1848
esp_console_repl_t (C++ *type*), 1849
esp_console_run (C++ *function*), 1844
esp_console_split_argv (C++ *function*), 1845

esp_console_start_repl (C++ function), 1846
esp_cpu_clear_breakpoint (C++ function), 2092
esp_cpu_clear_watchpoint (C++ function), 2092
esp_cpu_compare_and_set (C++ function), 2093
esp_cpu_configure_region_protection (C++ function), 2092
esp_cpu_cycle_count_t (C++ type), 2094
esp_cpu_dbgr_break (C++ function), 2093
esp_cpu_dbgr_is_attached (C++ function), 2093
esp_cpu_get_call_addr (C++ function), 2093
esp_cpu_get_core_id (C++ function), 2090
esp_cpu_get_cycle_count (C++ function), 2090
esp_cpu_get_sp (C++ function), 2090
ESP_CPU_INTR_DESC_FLAG_RESVD (C macro), 2094
ESP_CPU_INTR_DESC_FLAG_SPECIAL (C macro), 2094
esp_cpu_intr_desc_t (C++ struct), 2093
esp_cpu_intr_desc_t::flags (C++ member), 2093
esp_cpu_intr_desc_t::priority (C++ member), 2093
esp_cpu_intr_desc_t::type (C++ member), 2093
esp_cpu_intr_disable (C++ function), 2091
esp_cpu_intr_edge_ack (C++ function), 2091
esp_cpu_intr_enable (C++ function), 2091
esp_cpu_intr_get_desc (C++ function), 2090
esp_cpu_intr_get_enabled_mask (C++ function), 2091
esp_cpu_intr_get_handler_arg (C++ function), 2091
esp_cpu_intr_handler_t (C++ type), 2094
esp_cpu_intr_has_handler (C++ function), 2091
esp_cpu_intr_set_handler (C++ function), 2091
esp_cpu_intr_set_ivt_addr (C++ function), 2091
esp_cpu_intr_type_t (C++ enum), 2094
esp_cpu_intr_type_t::ESP_CPU_INTR_TYPE_EDGE (C++ enumerator), 2094
esp_cpu_intr_type_t::ESP_CPU_INTR_TYPE_LEVEL (C++ enumerator), 2094
esp_cpu_intr_type_t::ESP_CPU_INTR_TYPE_NA (C++ enumerator), 2094
esp_cpu_pc_to_addr (C++ function), 2090
esp_cpu_reset (C++ function), 2090
esp_cpu_set_breakpoint (C++ function), 2092
esp_cpu_set_cycle_count (C++ function), 2090
esp_cpu_set_watchpoint (C++ function), 2092
esp_cpu_stall (C++ function), 2090
esp_cpu_unstall (C++ function), 2090
esp_cpu_wait_for_intr (C++ function), 2090
esp_cpu_watchpoint_trigger_t (C++ enum), 2094
esp_cpu_watchpoint_trigger_t::ESP_CPU_WATCHPOINT_ (C++ enumerator), 2094
esp_cpu_watchpoint_trigger_t::ESP_CPU_WATCHPOINT_ (C++ enumerator), 2094
esp_cpu_watchpoint_trigger_t::ESP_CPU_WATCHPOINT_ (C++ enumerator), 2094
esp_crt_bundle_attach (C++ function), 197
esp_crt_bundle_detach (C++ function), 197
esp_crt_bundle_set (C++ function), 197
esp_deep_sleep (C++ function), 2131
esp_deep_sleep_disable_rom_logging (C++ function), 2132
esp_deep_sleep_start (C++ function), 2130
esp_deep_sleep_wake_stub_fn_t (C++ type), 2132
ESP_DEFAULT_GATT_IF (C macro), 237
esp_default_wake_deep_sleep (C++ function), 2131
esp_deregister_freertos_idle_hook (C++ function), 2030
esp_deregister_freertos_idle_hook_for_cpu (C++ function), 2030
esp_deregister_freertos_tick_hook (C++ function), 2030
esp_deregister_freertos_tick_hook_for_cpu (C++ function), 2030
esp_derive_local_mac (C++ function), 2087
ESP_DPP_AUTH_TIMEOUT_SECS (C macro), 942
ESP_DRAM_LOGD (C macro), 2080
ESP_DRAM_LOGE (C macro), 2079
ESP_DRAM_LOGI (C macro), 2080
ESP_DRAM_LOGV (C macro), 2080
ESP_DRAM_LOGW (C macro), 2080
esp_duplicate_info_t (C++ type), 303
esp_duplicate_scan_exceptional_list_type_t (C++ enum), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_duplicate_scan_exceptional_list_type_t::ESP_B (C++ enumerator), 316
esp_eap_client_clear_ca_cert (C++ function), 932

esp_eap_client_clear_certificate_and_key (C++ function), 932
esp_eap_client_clear_identity (C++ function), 931
esp_eap_client_clear_new_password (C++ function), 932
esp_eap_client_clear_password (C++ function), 931
esp_eap_client_clear_username (C++ function), 931
esp_eap_client_get_disable_time_check (C++ function), 933
esp_eap_client_set_ca_cert (C++ function), 932
esp_eap_client_set_certificate_and_key (C++ function), 932
esp_eap_client_set_disable_time_check (C++ function), 932
esp_eap_client_set_fast_params (C++ function), 933
esp_eap_client_set_identity (C++ function), 930
esp_eap_client_set_new_password (C++ function), 931
esp_eap_client_set_pac_file (C++ function), 933
esp_eap_client_set_password (C++ function), 931
esp_eap_client_set_suiteb_192bit_certification (C++ function), 933
esp_eap_client_set_ttls_phase2_method (C++ function), 933
esp_eap_client_set_username (C++ function), 931
esp_eap_client_use_default_cert_bundle (C++ function), 934
esp_eap_fast_config (C++ struct), 934
esp_eap_fast_config::fast_max_pac_list_len (C++ member), 934
esp_eap_fast_config::fast_pac_format_binary (C++ member), 934
esp_eap_fast_config::fast_provisioning (C++ member), 934
esp_eap_ttls_phase2_types (C++ enum), 934
esp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_CHAP (C++ enumerator), 935
esp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_EAP (C++ enumerator), 934
esp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_MSCHAP (C++ enumerator), 934
esp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_MSCHAPV2 (C++ enumerator), 934
esp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_PAP (C++ enumerator), 934
ESP_EARLY_LOGD (C macro), 2078
ESP_EARLY_LOGE (C macro), 2078
ESP_EARLY_LOGI (C macro), 2078
ESP_EARLY_LOGV (C macro), 2078
ESP_EARLY_LOGW (C macro), 2078
esp_efuse_batch_write_begin (C++ function), 1865
esp_efuse_batch_write_cancel (C++ function), 1866
esp_efuse_batch_write_commit (C++ function), 1866
esp_efuse_block_is_empty (C++ function), 1866
esp_efuse_block_t (C++ enum), 1858
esp_efuse_block_t::EFUSE_BLK0 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK1 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK2 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK3 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_ENCRYPT_FLASH (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_KEY0 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_KEY1 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_KEY2 (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_KEY_MAX (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_MAX (C++ enumerator), 1859
esp_efuse_block_t::EFUSE_BLK_SECURE_BOOT (C++ enumerator), 1859
esp_efuse_check_errors (C++ function), 1868
esp_efuse_check_secure_version (C++ function), 1864
esp_efuse_coding_scheme_t (C++ enum), 1859
esp_efuse_coding_scheme_t::EFUSE_CODING_SCHEME_3_4 (C++ enumerator), 1859
esp_efuse_coding_scheme_t::EFUSE_CODING_SCHEME_NONE (C++ enumerator), 1859
esp_efuse_coding_scheme_t::EFUSE_CODING_SCHEME_RS (C++ enumerator), 1859
esp_efuse_desc_t (C++ struct), 1868
esp_efuse_desc_t::bit_count (C++ member), 1868
esp_efuse_desc_t::bit_start (C++ member), 1868
esp_efuse_desc_t::efuse_block (C++ member), 1868
esp_efuse_disable_basic_rom_console (C++ function), 1864
esp_efuse_disable_rom_download_mode (C++ function), 1864
esp_efuse_find_purpose (C++ function), 1867
esp_efuse_get_coding_scheme (C++ function), 1863
esp_efuse_get_field_size (C++ function),

1862
esp_efuse_get_key_dis_read (C++ function), 1866
esp_efuse_get_key_dis_write (C++ function), 1866
esp_efuse_get_key_purpose (C++ function), 1867
esp_efuse_get_keypurpose_dis_write (C++ function), 1867
esp_efuse_get_pkg_ver (C++ function), 1863
esp_efuse_key_block_unused (C++ function), 1867
esp_efuse_mac_get_custom (C++ function), 2087
esp_efuse_mac_get_default (C++ function), 2087
esp_efuse_purpose_t (C++ enum), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_UP (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_MAX (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_V2 (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_SYS_TEST (C++ enumerator), 1860
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_USER (C++ enumerator), 1860
esp_efuse_read_block (C++ function), 1863
esp_efuse_read_field_bit (C++ function), 1860
esp_efuse_read_field_blob (C++ function), 1860
esp_efuse_read_field_cnt (C++ function), 1861
esp_efuse_read_reg (C++ function), 1862
esp_efuse_read_secure_version (C++ function), 1864
esp_efuse_reset (C++ function), 1863
esp_efuse_rom_log_scheme_t (C++ enum), 1869
esp_efuse_rom_log_scheme_t::ESP_EFUSE_ROM_LOG_ALWAYS_OFF (C++ enumerator), 1869
esp_efuse_rom_log_scheme_t::ESP_EFUSE_ROM_LOG_ALWAYS_ON (C++ enumerator), 1869
esp_efuse_rom_log_scheme_t::ESP_EFUSE_ROM_LOG_CONTROLLED_BY_GPIO_HIGH (C++ enumerator), 1869
esp_efuse_rom_log_scheme_t::ESP_EFUSE_ROM_LOG_CONTROLLED_BY_GPIO_LOW (C++ enumerator), 1869
esp_efuse_set_key_dis_read (C++ function), 1866
esp_efuse_set_key_dis_write (C++ function), 1866
esp_efuse_set_read_protect (C++ function), 1862
esp_efuse_set_rom_log_scheme (C++ function), 1864
esp_efuse_set_write_protect (C++ function), 1862
esp_efuse_update_secure_version (C++ function), 1865
esp_efuse_write_block (C++ function), 1863
esp_efuse_write_field_bit (C++ function), 1862
esp_efuse_write_field_blob (C++ function), 1861
esp_efuse_write_field_cnt (C++ function), 1861
esp_efuse_write_key (C++ function), 1867
esp_efuse_write_keys (C++ function), 1868
esp_efuse_write_reg (C++ function), 1862
ESP_ERR_CODING (C macro), 1869
ESP_ERR_DAMAGED_READING (C macro), 1869
ESP_ERR_DPP_AUTH_TIMEOUT (C macro), 943
ESP_ERR_DPP_FAILURE (C macro), 942
ESP_ERR_DPP_INVALID_ATTR (C macro), 943
ESP_ERR_DPP_MSG_DISPATCH (C macro), 943
ESP_ERR_EFUSE (C macro), 1869
ESP_ERR_EFUSE_CNT_IS_FULL (C macro), 1869
ESP_ERR_EFUSE_REPEATED_PROG (C macro), 1869
ESP_ERR_ESP_NETIF_BASE (C macro), 1009
ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED (C macro), 1009
ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED (C macro), 1009
ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED (C macro), 1009
ESP_ERR_ESP_NETIF_DHCPC_START_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_DHCPS_START_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED (C macro), 1009
ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_IF_NOT_READY (C macro), 1009
ESP_ERR_ESP_NETIF_INIT_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_INVALID_PARAMS (C macro), 1009
ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_MLD6_FAILED (C macro), 1009
ESP_ERR_ESP_NETIF_NO_MEM (C macro), 1009
ESP_ERR_ESP_TLS_BASE (C macro), 152
ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET (C macro), 152
ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME (C macro), 152
ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT (C macro), 153
ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST (C macro), 152
ESP_ERR_ESP_TLS_SE_FAILED (C macro), 153

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED (*C macro*), 153
ESP_ERR_ESP_TLS_TCP_CLOSED_FIN (*C macro*), 153
ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY (*C macro*), 152
ESP_ERR_ESPNOW_ARG (*C macro*), 829
ESP_ERR_ESPNOW_BASE (*C macro*), 829
ESP_ERR_ESPNOW_EXIST (*C macro*), 829
ESP_ERR_ESPNOW_FULL (*C macro*), 829
ESP_ERR_ESPNOW_IF (*C macro*), 829
ESP_ERR_ESPNOW_INTERNAL (*C macro*), 829
ESP_ERR_ESPNOW_NO_MEM (*C macro*), 829
ESP_ERR_ESPNOW_NOT_FOUND (*C macro*), 829
ESP_ERR_ESPNOW_NOT_INIT (*C macro*), 829
ESP_ERR_FLASH_BASE (*C macro*), 1872
ESP_ERR_FLASH_NOT_INITIALISED (*C macro*), 1792
ESP_ERR_FLASH_OP_FAIL (*C macro*), 1785
ESP_ERR_FLASH_OP_TIMEOUT (*C macro*), 1785
ESP_ERR_FLASH_PROTECTED (*C macro*), 1792
ESP_ERR_FLASH_UNSUPPORTED_CHIP (*C macro*), 1792
ESP_ERR_FLASH_UNSUPPORTED_HOST (*C macro*), 1792
ESP_ERR_HTTP_BASE (*C macro*), 166
ESP_ERR_HTTP_CONNECT (*C macro*), 166
ESP_ERR_HTTP_CONNECTING (*C macro*), 166
ESP_ERR_HTTP_CONNECTION_CLOSED (*C macro*), 166
ESP_ERR_HTTP_EAGAIN (*C macro*), 166
ESP_ERR_HTTP_FETCH_HEADER (*C macro*), 166
ESP_ERR_HTTP_INVALID_TRANSPORT (*C macro*), 166
ESP_ERR_HTTP_MAX_REDIRECT (*C macro*), 166
ESP_ERR_HTTP_WRITE_DATA (*C macro*), 166
ESP_ERR_HTTPD_ALLOC_MEM (*C macro*), 221
ESP_ERR_HTTPD_BASE (*C macro*), 221
ESP_ERR_HTTPD_HANDLER_EXISTS (*C macro*), 221
ESP_ERR_HTTPD_HANDLERS_FULL (*C macro*), 221
ESP_ERR_HTTPD_INVALID_REQ (*C macro*), 221
ESP_ERR_HTTPD_RESP_HDR (*C macro*), 221
ESP_ERR_HTTPD_RESP_SEND (*C macro*), 221
ESP_ERR_HTTPD_RESULT_TRUNC (*C macro*), 221
ESP_ERR_HTTPD_TASK (*C macro*), 221
ESP_ERR_HTTPS_OTA_BASE (*C macro*), 1878
ESP_ERR_HTTPS_OTA_IN_PROGRESS (*C macro*), 1878
ESP_ERR_HW_CRYPTO_BASE (*C macro*), 1872
ESP_ERR_INVALID_ARG (*C macro*), 1871
ESP_ERR_INVALID_CRC (*C macro*), 1871
ESP_ERR_INVALID_MAC (*C macro*), 1871
ESP_ERR_INVALID_RESPONSE (*C macro*), 1871
ESP_ERR_INVALID_SIZE (*C macro*), 1871
ESP_ERR_INVALID_STATE (*C macro*), 1871
ESP_ERR_INVALID_VERSION (*C macro*), 1871
ESP_ERR_MBEDTLS_CERT_PARTLY_OK (*C macro*), 153
ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_SETUP_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_SSL_WRITE_FAILED (*C macro*), 153
ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED (*C macro*), 153
ESP_ERR_MEMPROT_BASE (*C macro*), 1872
ESP_ERR_MESH_ARGUMENT (*C macro*), 862
ESP_ERR_MESH_BASE (*C macro*), 1872
ESP_ERR_MESH_DISCARD (*C macro*), 863
ESP_ERR_MESH_DISCARD_DUPLICATE (*C macro*), 863
ESP_ERR_MESH_DISCONNECTED (*C macro*), 863
ESP_ERR_MESH_EXCEED_MTU (*C macro*), 863
ESP_ERR_MESH_INTERFACE (*C macro*), 863
ESP_ERR_MESH_NO_MEMORY (*C macro*), 862
ESP_ERR_MESH_NO_PARENT_FOUND (*C macro*), 863
ESP_ERR_MESH_NO_ROUTE_FOUND (*C macro*), 863
ESP_ERR_MESH_NOT_ALLOWED (*C macro*), 862
ESP_ERR_MESH_NOT_CONFIG (*C macro*), 862
ESP_ERR_MESH_NOT_INIT (*C macro*), 862
ESP_ERR_MESH_NOT_START (*C macro*), 862
ESP_ERR_MESH_NOT_SUPPORT (*C macro*), 862
ESP_ERR_MESH_OPTION_NULL (*C macro*), 863
ESP_ERR_MESH_OPTION_UNKNOWN (*C macro*), 863
ESP_ERR_MESH_PS (*C macro*), 864
ESP_ERR_MESH_QUEUE_FAIL (*C macro*), 863
ESP_ERR_MESH_QUEUE_FULL (*C macro*), 863
ESP_ERR_MESH_QUEUE_READ (*C macro*), 863
ESP_ERR_MESH_RECV_RELEASE (*C macro*), 864
ESP_ERR_MESH_TIMEOUT (*C macro*), 863
ESP_ERR_MESH_VOTING (*C macro*), 863
ESP_ERR_MESH_WIFI_NOT_START (*C macro*), 862
ESP_ERR_MESH_XMIT (*C macro*), 863

ESP_ERR_MESH_XON_NO_WINDOW (*C macro*), 863
ESP_ERR_NO_MEM (*C macro*), 1871
ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS (*C macro*), 1869
ESP_ERR_NOT_FINISHED (*C macro*), 1871
ESP_ERR_NOT_FOUND (*C macro*), 1871
ESP_ERR_NOT_SUPPORTED (*C macro*), 1871
ESP_ERR_NVS_BASE (*C macro*), 1745
ESP_ERR_NVS_CONTENT_DIFFERS (*C macro*), 1747
ESP_ERR_NVS_CORRUPT_KEY_PART (*C macro*), 1747
ESP_ERR_NVS_ENCR_NOT_SUPPORTED (*C macro*), 1747
ESP_ERR_NVS_INVALID_HANDLE (*C macro*), 1746
ESP_ERR_NVS_INVALID_LENGTH (*C macro*), 1746
ESP_ERR_NVS_INVALID_NAME (*C macro*), 1746
ESP_ERR_NVS_INVALID_STATE (*C macro*), 1746
ESP_ERR_NVS_KEY_TOO_LONG (*C macro*), 1746
ESP_ERR_NVS_KEYS_NOT_INITIALIZED (*C macro*), 1747
ESP_ERR_NVS_NEW_VERSION_FOUND (*C macro*), 1746
ESP_ERR_NVS_NO_FREE_PAGES (*C macro*), 1746
ESP_ERR_NVS_NOT_ENOUGH_SPACE (*C macro*), 1746
ESP_ERR_NVS_NOT_FOUND (*C macro*), 1745
ESP_ERR_NVS_NOT_INITIALIZED (*C macro*), 1745
ESP_ERR_NVS_PAGE_FULL (*C macro*), 1746
ESP_ERR_NVS_PART_NOT_FOUND (*C macro*), 1746
ESP_ERR_NVS_READ_ONLY (*C macro*), 1745
ESP_ERR_NVS_REMOVE_FAILED (*C macro*), 1746
ESP_ERR_NVS_TYPE_MISMATCH (*C macro*), 1745
ESP_ERR_NVS_VALUE_TOO_LONG (*C macro*), 1746
ESP_ERR_NVS_WRONG_ENCRYPTION (*C macro*), 1747
ESP_ERR_NVS_XTS_CFG_FAILED (*C macro*), 1746
ESP_ERR_NVS_XTS_CFG_NOT_FOUND (*C macro*), 1746
ESP_ERR_NVS_XTS_DECR_FAILED (*C macro*), 1746
ESP_ERR_NVS_XTS_ENCR_FAILED (*C macro*), 1746
ESP_ERR_OTA_BASE (*C macro*), 2106
ESP_ERR_OTA_PARTITION_CONFLICT (*C macro*), 2106
ESP_ERR_OTA_ROLLBACK_FAILED (*C macro*), 2106
ESP_ERR_OTA_ROLLBACK_INVALID_STATE (*C macro*), 2106
ESP_ERR_OTA_SELECT_INFO_INVALID (*C macro*), 2106
ESP_ERR_OTA_SMALL_SEC_VER (*C macro*), 2106
ESP_ERR_OTA_VALIDATE_FAILED (*C macro*), 2106
esp_err_t (*C++ type*), 1872
ESP_ERR_TIMEOUT (*C macro*), 1871
esp_err_to_name (*C++ function*), 1870
esp_err_to_name_r (*C++ function*), 1870
ESP_ERR_ULP_BASE (*C macro*), 2182
ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (*C macro*), 2183
ESP_ERR_ULP_DUPLICATE_LABEL (*C macro*), 2182
ESP_ERR_ULP_INVALID_LOAD_ADDR (*C macro*), 2182
ESP_ERR_ULP_SIZE_TOO_BIG (*C macro*), 2182
ESP_ERR_ULP_UNDEFINED_LABEL (*C macro*), 2183
ESP_ERR_WIFI_BASE (*C macro*), 1872
ESP_ERR_WIFI_CONN (*C macro*), 893
ESP_ERR_WIFI_DISCARD (*C macro*), 894
ESP_ERR_WIFI_IF (*C macro*), 893
ESP_ERR_WIFI_INIT_STATE (*C macro*), 894
ESP_ERR_WIFI_MAC (*C macro*), 893
ESP_ERR_WIFI_MODE (*C macro*), 893
ESP_ERR_WIFI_NOT_ASSOC (*C macro*), 894
ESP_ERR_WIFI_NOT_CONNECT (*C macro*), 894
ESP_ERR_WIFI_NOT_INIT (*C macro*), 893
ESP_ERR_WIFI_NOT_STARTED (*C macro*), 893
ESP_ERR_WIFI_NOT_STOPPED (*C macro*), 893
ESP_ERR_WIFI_NVS (*C macro*), 893
ESP_ERR_WIFI_PASSWORD (*C macro*), 894
ESP_ERR_WIFI_POST (*C macro*), 894
ESP_ERR_WIFI_REGISTRAR (*C macro*), 937
ESP_ERR_WIFI_ROC_IN_PROGRESS (*C macro*), 894
ESP_ERR_WIFI_SSID (*C macro*), 894
ESP_ERR_WIFI_STATE (*C macro*), 893
ESP_ERR_WIFI_STOP_STATE (*C macro*), 894
ESP_ERR_WIFI_TIMEOUT (*C macro*), 894
ESP_ERR_WIFI_TX_DISALLOW (*C macro*), 894
ESP_ERR_WIFI_WAKE_FAIL (*C macro*), 894
ESP_ERR_WIFI_WOULD_BLOCK (*C macro*), 894
ESP_ERR_WIFI_WPS_SM (*C macro*), 937
ESP_ERR_WIFI_WPS_TYPE (*C macro*), 937
ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_CTX_SETUP_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_SSL_SETUP_FAILED (*C macro*), 154
ESP_ERR_WOLFSSL_SSL_WRITE_FAILED (*C macro*), 154
ESP_ERROR_CHECK (*C macro*), 1872
ESP_ERROR_CHECK_WITHOUT_ABORT (*C macro*), 1872

esp_esptouch_set_timeout (C++ function), 871
esp_eth_config_t (C++ struct), 957
esp_eth_config_t::check_link_period_ms (C++ member), 957
esp_eth_config_t::mac (C++ member), 957
esp_eth_config_t::on_lowlevel_deinit_done (C++ member), 957
esp_eth_config_t::on_lowlevel_init_done (C++ member), 957
esp_eth_config_t::phy (C++ member), 957
esp_eth_config_t::read_phy_reg (C++ member), 958
esp_eth_config_t::stack_input (C++ member), 957
esp_eth_config_t::write_phy_reg (C++ member), 958
esp_eth_decrease_reference (C++ function), 957
esp_eth_del_netif_glue (C++ function), 980
esp_eth_driver_install (C++ function), 954
esp_eth_driver_uninstall (C++ function), 954
esp_eth_handle_t (C++ type), 959
esp_eth_increase_reference (C++ function), 956
esp_eth_io_cmd_t (C++ enum), 959
esp_eth_io_cmd_t::ETH_CMD_CUSTOM_MAC_CMDS (C++ enumerator), 960
esp_eth_io_cmd_t::ETH_CMD_CUSTOM_PHY_CMDS (C++ enumerator), 960
esp_eth_io_cmd_t::ETH_CMD_G_AUTONEGO (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_G_DUPLEX_MODE (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_G_MAC_ADDR (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_G_PHY_ADDR (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_G_SPEED (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_READ_PHY_REG (C++ enumerator), 960
esp_eth_io_cmd_t::ETH_CMD_S_AUTONEGO (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_S_DUPLEX_MODE (C++ enumerator), 960
esp_eth_io_cmd_t::ETH_CMD_S_FLOW_CTRL (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_S_MAC_ADDR (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_S_PHY_ADDR (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_S_PHY_LOOPBACK (C++ enumerator), 960
esp_eth_io_cmd_t::ETH_CMD_S_PROMISCUOUS (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_S_SPEED (C++ enumerator), 959
esp_eth_io_cmd_t::ETH_CMD_WRITE_PHY_REG (C++ enumerator), 960
esp_eth_ioctl (C++ function), 956
esp_eth_mac_new_esp32 (C++ function), 962
esp_eth_mac_s (C++ struct), 963
esp_eth_mac_s::custom_ioctl (C++ member), 966
esp_eth_mac_s::deinit (C++ member), 963
esp_eth_mac_s::del (C++ member), 967
esp_eth_mac_s::enable_flow_ctrl (C++ member), 966
esp_eth_mac_s::get_addr (C++ member), 965
esp_eth_mac_s::init (C++ member), 963
esp_eth_mac_s::read_phy_reg (C++ member), 964
esp_eth_mac_s::receive (C++ member), 964
esp_eth_mac_s::set_addr (C++ member), 965
esp_eth_mac_s::set_duplex (C++ member), 965
esp_eth_mac_s::set_link (C++ member), 966
esp_eth_mac_s::set_mediator (C++ member), 963
esp_eth_mac_s::set_peer_pause_ability (C++ member), 966
esp_eth_mac_s::set_promiscuous (C++ member), 966
esp_eth_mac_s::set_speed (C++ member), 965
esp_eth_mac_s::start (C++ member), 963
esp_eth_mac_s::stop (C++ member), 963
esp_eth_mac_s::transmit (C++ member), 963
esp_eth_mac_s::transmit_vargs (C++ member), 964
esp_eth_mac_s::write_phy_reg (C++ member), 965
esp_eth_mac_t (C++ type), 968
esp_eth_mediator_s (C++ struct), 960
esp_eth_mediator_s::on_state_changed (C++ member), 961
esp_eth_mediator_s::phy_reg_read (C++ member), 960
esp_eth_mediator_s::phy_reg_write (C++ member), 960
esp_eth_mediator_s::stack_input (C++ member), 961
esp_eth_mediator_t (C++ type), 961
esp_eth_netif_glue_handle_t (C++ type), 980
esp_eth_new_netif_glue (C++ function), 980
esp_eth_phy_802_3_advertise_pause_ability (C++ function), 975
esp_eth_phy_802_3_autonego_ctrl (C++ function), 974
esp_eth_phy_802_3_basic_phy_deinit (C++ function), 977
esp_eth_phy_802_3_basic_phy_init (C++ function), 976

esp_eth_phy_802_3_deinit (C++ function), 976
esp_eth_phy_802_3_del (C++ function), 976
esp_eth_phy_802_3_detect_phy_addr (C++ function), 976
esp_eth_phy_802_3_get_addr (C++ function), 975
esp_eth_phy_802_3_get_mmd_addr (C++ function), 977
esp_eth_phy_802_3_init (C++ function), 976
esp_eth_phy_802_3_loopback (C++ function), 975
esp_eth_phy_802_3_mmd_func_t (C++ enum), 980
esp_eth_phy_802_3_mmd_func_t::MMD_FUNC_ADDRESS (C++ enumerator), 980
esp_eth_phy_802_3_mmd_func_t::MMD_FUNC_DATA_NO_INCR (C++ enumerator), 980
esp_eth_phy_802_3_mmd_func_t::MMD_FUNC_DATA_RW_INCR (C++ enumerator), 980
esp_eth_phy_802_3_mmd_func_t::MMD_FUNC_DATA_WR_INCR (C++ enumerator), 980
esp_eth_phy_802_3_obj_config_init (C++ function), 979
esp_eth_phy_802_3_pwrctl (C++ function), 974
esp_eth_phy_802_3_read_manufac_info (C++ function), 977
esp_eth_phy_802_3_read_mmd_data (C++ function), 978
esp_eth_phy_802_3_read_mmd_register (C++ function), 978
esp_eth_phy_802_3_read_oui (C++ function), 977
esp_eth_phy_802_3_reset (C++ function), 974
esp_eth_phy_802_3_reset_hw (C++ function), 976
esp_eth_phy_802_3_set_addr (C++ function), 975
esp_eth_phy_802_3_set_duplex (C++ function), 975
esp_eth_phy_802_3_set_link (C++ function), 975
esp_eth_phy_802_3_set_mediator (C++ function), 974
esp_eth_phy_802_3_set_mmd_addr (C++ function), 977
esp_eth_phy_802_3_set_speed (C++ function), 975
esp_eth_phy_802_3_write_mmd_data (C++ function), 978
esp_eth_phy_802_3_write_mmd_register (C++ function), 978
ESP_ETH_PHY_ADDR_AUTO (C macro), 973
esp_eth_phy_into_phy_802_3 (C++ function), 979
esp_eth_phy_new_dp83848 (C++ function), 969
esp_eth_phy_new_ip101 (C++ function), 969
esp_eth_phy_new_ksz80xx (C++ function), 969
esp_eth_phy_new_lan87xx (C++ function), 969
esp_eth_phy_new_rtl8201 (C++ function), 969
esp_eth_phy_reg_rw_data_t (C++ struct), 958
esp_eth_phy_reg_rw_data_t::reg_addr (C++ member), 958
esp_eth_phy_reg_rw_data_t::reg_value_p (C++ member), 958
esp_eth_phy_s (C++ struct), 970
esp_eth_phy_s::advertise_pause_ability (C++ member), 972
esp_eth_phy_s::autonego_ctrl (C++ member), 971
esp_eth_phy_s::custom_ioctl (C++ member), 972
esp_eth_phy_s::deinit (C++ member), 970
esp_eth_phy_s::del (C++ member), 973
esp_eth_phy_s::get_addr (C++ member), 971
esp_eth_phy_s::get_link (C++ member), 971
esp_eth_phy_s::init (C++ member), 970
esp_eth_phy_s::loopback (C++ member), 972
esp_eth_phy_s::pwrctl (C++ member), 971
esp_eth_phy_s::reset (C++ member), 970
esp_eth_phy_s::reset_hw (C++ member), 970
esp_eth_phy_s::set_addr (C++ member), 971
esp_eth_phy_s::set_duplex (C++ member), 972
esp_eth_phy_s::set_link (C++ member), 971
esp_eth_phy_s::set_mediator (C++ member), 970
esp_eth_phy_s::set_speed (C++ member), 972
esp_eth_phy_t (C++ type), 973
esp_eth_start (C++ function), 954
esp_eth_state_t (C++ enum), 961
esp_eth_state_t::ETH_STATE_DEINIT (C++ enumerator), 961
esp_eth_state_t::ETH_STATE_DUPLEX (C++ enumerator), 961
esp_eth_state_t::ETH_STATE_LINK (C++ enumerator), 961
esp_eth_state_t::ETH_STATE_LLINIT (C++ enumerator), 961
esp_eth_state_t::ETH_STATE_PAUSE (C++ enumerator), 961
esp_eth_state_t::ETH_STATE_SPEED (C++ enumerator), 961
esp_eth_stop (C++ function), 954
esp_eth_transmit (C++ function), 955
esp_eth_transmit_vargs (C++ function), 955
esp_eth_update_input_path (C++ function), 955
ESP_EVENT_ANY_BASE (C macro), 1891
ESP_EVENT_ANY_ID (C macro), 1891
ESP_EVENT_DECLARE_BASE (C macro), 1891
ESP_EVENT_DEFINE_BASE (C macro), 1891
esp_event_dump (C++ function), 1890
esp_event_handler_instance_register

(C++ function), 1886
esp_event_handler_instance_register_with (C++ function), 1885
esp_event_handler_instance_t (C++ type), 1891
esp_event_handler_instance_unregister (C++ function), 1888
esp_event_handler_instance_unregister_with (C++ function), 1887
esp_event_handler_register (C++ function), 1884
esp_event_handler_register_with (C++ function), 1884
esp_event_handler_t (C++ type), 1891
esp_event_handler_unregister (C++ function), 1886
esp_event_handler_unregister_with (C++ function), 1887
esp_event_isr_post (C++ function), 1889
esp_event_isr_post_to (C++ function), 1889
esp_event_loop_args_t (C++ struct), 1890
esp_event_loop_args_t::queue_size (C++ member), 1891
esp_event_loop_args_t::task_core_id (C++ member), 1891
esp_event_loop_args_t::task_name (C++ member), 1891
esp_event_loop_args_t::task_priority (C++ member), 1891
esp_event_loop_args_t::task_stack_size (C++ member), 1891
esp_event_loop_create (C++ function), 1883
esp_event_loop_create_default (C++ function), 1883
esp_event_loop_delete (C++ function), 1883
esp_event_loop_delete_default (C++ function), 1883
esp_event_loop_handle_t (C++ type), 1891
esp_event_loop_run (C++ function), 1883
esp_event_post (C++ function), 1888
esp_event_post_to (C++ function), 1888
ESP_EXECUTE_EXPRESSION_WITH_STACK (C macro), 1837
esp_execute_shared_stack_function (C++ function), 1837
ESP_FAIL (C macro), 1871
esp_fill_random (C++ function), 2121
esp_flash_chip_driver_initialized (C++ function), 1776
esp_flash_enc_mode_t (C++ enum), 1803
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_DEVELOPMENT (C++ enumerator), 1803
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_DISABLED (C++ enumerator), 1803
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_RELEASE (C++ enumerator), 1803
esp_flash_encrypt_check_and_update (C++ function), 1801
esp_flash_encrypt_contents (C++ function), 1801
esp_flash_encrypt_enable (C++ function), 1802
esp_flash_encrypt_init (C++ function), 1801
esp_flash_encrypt_initialized_once (C++ function), 1801
esp_flash_encrypt_is_write_protected (C++ function), 1802
esp_flash_encrypt_region (C++ function), 1802
esp_flash_encrypt_state (C++ function), 1801
esp_flash_encryption_cfg_verify_release_mode (C++ function), 1803
esp_flash_encryption_enable_secure_features (C++ function), 1803
esp_flash_encryption_enabled (C++ function), 1801
esp_flash_encryption_init_checks (C++ function), 1802
esp_flash_encryption_set_release_mode (C++ function), 1803
esp_flash_erase_chip (C++ function), 1778
esp_flash_erase_region (C++ function), 1778
esp_flash_get_chip_write_protect (C++ function), 1778
esp_flash_get_physical_size (C++ function), 1777
esp_flash_get_protectable_regions (C++ function), 1779
esp_flash_get_protected_region (C++ function), 1779
esp_flash_get_size (C++ function), 1777
esp_flash_init (C++ function), 1776
esp_flash_io_mode_t (C++ enum), 1791
esp_flash_io_mode_t::SPI_FLASH_DIO (C++ enumerator), 1791
esp_flash_io_mode_t::SPI_FLASH_DOUT (C++ enumerator), 1791
esp_flash_io_mode_t::SPI_FLASH_FASTRD (C++ enumerator), 1791
esp_flash_io_mode_t::SPI_FLASH_OPI_DTR (C++ enumerator), 1792
esp_flash_io_mode_t::SPI_FLASH_OPI_STR (C++ enumerator), 1792
esp_flash_io_mode_t::SPI_FLASH_QIO (C++ enumerator), 1792
esp_flash_io_mode_t::SPI_FLASH_QOUT (C++ enumerator), 1792
esp_flash_io_mode_t::SPI_FLASH_READ_MODE_MAX (C++ enumerator), 1792
esp_flash_io_mode_t::SPI_FLASH_SLOWRD (C++ enumerator), 1791
esp_flash_is_quad_mode (C++ function), 1781
esp_flash_os_functions_t (C++ struct), 1781
esp_flash_os_functions_t::check_yield (C++ member), 1782

esp_flash_os_functions_t::delay_us (C++ member), 1782
esp_flash_os_functions_t::end (C++ member), 1782
esp_flash_os_functions_t::get_system_time (C++ member), 1782
esp_flash_os_functions_t::get_temp_buffer (C++ member), 1782
esp_flash_os_functions_t::region_protected (C++ member), 1782
esp_flash_os_functions_t::release_temp_buffer (C++ member), 1782
esp_flash_os_functions_t::set_flash_op_status (C++ member), 1782
esp_flash_os_functions_t::start (C++ member), 1782
esp_flash_os_functions_t::yield (C++ member), 1782
esp_flash_read (C++ function), 1780
esp_flash_read_encrypted (C++ function), 1781
esp_flash_read_id (C++ function), 1776
esp_flash_read_unique_chip_id (C++ function), 1777
esp_flash_region_t (C++ struct), 1781
esp_flash_region_t::offset (C++ member), 1781
esp_flash_region_t::size (C++ member), 1781
esp_flash_set_chip_write_protect (C++ function), 1778
esp_flash_set_protected_region (C++ function), 1779
esp_flash_speed_s (C++ enum), 1791
esp_flash_speed_s::ESP_FLASH_10MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_120MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_20MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_26MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_40MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_5MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_80MHZ (C++ enumerator), 1791
esp_flash_speed_s::ESP_FLASH_SPEED_MAX (C++ enumerator), 1791
esp_flash_speed_t (C++ type), 1790
esp_flash_spi_device_config_t (C++ struct), 1775
esp_flash_spi_device_config_t::cs_id (C++ member), 1776
esp_flash_spi_device_config_t::cs_io_num (C++ member), 1776
esp_flash_spi_device_config_t::freq_mhz (C++ member), 1776
esp_flash_spi_device_config_t::host_id (C++ member), 1775
esp_flash_spi_device_config_t::input_delay_ns (C++ member), 1776
esp_flash_spi_device_config_t::io_mode (C++ member), 1776
esp_flash_spi_device_config_t::speed (C++ member), 1776
esp_flash_t (C++ struct), 1782
esp_flash_t (C++ type), 1783
esp_flash_t::busy (C++ member), 1783
esp_flash_t::chip_drv (C++ member), 1782
esp_flash_t::chip_id (C++ member), 1783
esp_flash_t::host (C++ member), 1782
esp_flash_t::hpm_dummy_ena (C++ member), 1783
esp_flash_t::os_func (C++ member), 1783
esp_flash_t::os_func_data (C++ member), 1783
esp_flash_t::read_mode (C++ member), 1783
esp_flash_t::reserved_flags (C++ member), 1783
esp_flash_t::size (C++ member), 1783
esp_flash_write (C++ function), 1780
esp_flash_write_encrypted (C++ function), 1780
esp_flash_write_protect_crypt_cnt (C++ function), 1802
esp_freertos_idle_cb_t (C++ type), 2030
esp_freertos_tick_cb_t (C++ type), 2030
ESP_GAP_BLE_ADD_WHITELIST_COMPLETE_EVT (C macro), 299
esp_gap_ble_cb_event_t (C++ enum), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_CLEAR_COM (C++ enumerator), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_DATA_RAW_ (C++ enumerator), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_DATA_SET_ (C++ enumerator), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_START_COM (C++ enumerator), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_STOP_COMP (C++ enumerator), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_ADV_TERMINATE (C++ enumerator), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_AUTH_CMPL_EVT (C++ enumerator), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_CHANNEL_SELEC (C++ enumerator), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_CLEAR_BOND_DE (C++ enumerator), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_DTM_TEST_UPDA (C++ enumerator), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EVT_MAX (C++ enumerator), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_DATA_ (C++ enumerator), 306

esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_REPORT_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_SET_CLEAR_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_SET_PARAMS_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_SET_RAND_ADDR_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_SET_REMOVE_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_START_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_ADV_STOP_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_SCAN_RSP_DATA_SET_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_SCAN_START_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_EXT_SCAN_STOP_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_GET_DEV_NAME_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_KEY_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_LOCAL_ER_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_LOCAL_IR_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_NC_REQ_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_OOB_REQ_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PASSKEY_NOTIF_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PASSKEY_REQ_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_ADD_DEV_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_CLEAR_DEV_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_CREATE_SYNC_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_DATA_SET_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_RECV_ENABLE_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_REMOVE_DEV_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_SYNC_CANCEL_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_SYNC_ESTAB_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_SYNC_LOST_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_SYNC_TRANS_PARAMS_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_START_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PERIODIC_ADV_STOP_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PHY_UPDATE_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_PREFER_EXT_CONN_PARAMS_SET_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_READ_PHY_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SC_CR_LOC_OOB_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SC_OOB_REQ_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_REQ_RECEIVED_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_RESULT_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_START_COMPLETE_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SCAN_TIMEOUT_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SEC_REQ_EVT (*C++ enumerator*), 304
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_CHANNELS_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_EXT_SCAN_PARAM_COMPLETE_EVT (*C++ enumerator*), 307
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_PAST_PARAMS_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT (*C++ enumerator*), 306

esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_PREFERRED_DEFAULT_PHY_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_UPDATE_PERIPHER_CONN_PARAM_COMPLETE_EVT (*C++ enumerator*), 306
esp_gap_ble_cb_event_t::ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT (*C++ enumerator*), 305
esp_gap_ble_cb_event_t::ESP_GAP_BLE_VENDOR_CMD_COMPLETE_EVT (*C++ enumerator*), 308
esp_gap_ble_cb_t (*C++ type*), 304
esp_gap_ble_channels (*C++ type*), 303
ESP_GAP_BLE_CHANNELS_LEN (*C macro*), 299
esp_gap_ble_set_authorization (*C++ function*), 252
esp_gap_ble_set_channels (*C++ function*), 251
esp_gap_conn_params_t (*C++ struct*), 281
esp_gap_conn_params_t::interval (*C++ member*), 281
esp_gap_conn_params_t::latency (*C++ member*), 281
esp_gap_conn_params_t::timeout (*C++ member*), 281
esp_gap_search_evt_t (*C++ enum*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_DISC_CMPL_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_DISC_BLE_RES_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_DISC_CMPL_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_DISC_RES_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_INQ_CMPL_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_INQ_RES_EVT (*C++ enumerator*), 314
esp_gap_search_evt_t::ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT (*C++ enumerator*), 314
ESP_GATT_ATTR_HANDLE_MAX (*C macro*), 326
esp_gatt_auth_req_t (*C++ enum*), 330
esp_gatt_auth_req_t::ESP_GATT_AUTH_REQ_MITM (*C++ enumerator*), 331
esp_gatt_auth_req_t::ESP_GATT_AUTH_REQ_NO_MITM (*C++ enumerator*), 330
esp_gatt_auth_req_t::ESP_GATT_AUTH_REQ_NONE (*C++ enumerator*), 330
esp_gatt_auth_req_t::ESP_GATT_AUTH_REQ_SIGNED_MITM (*C++ enumerator*), 331
esp_gatt_auth_req_t::ESP_GATT_AUTH_REQ_SIGNED_NO_MITM (*C++ enumerator*), 331
ESP_GATT_AUTO_RSP (*C macro*), 327
ESP_GATT_BODY_SENSOR_LOCATION (*C macro*), 325
ESP_GATT_CHAR_PROP_BIT_BROADCAST (*C macro*), 327
ESP_GATT_CHAR_PROP_BIT_EXT_PROP (*C macro*), 327
ESP_GATT_CHAR_PROP_BIT_NOTIFY (*C macro*), 327
ESP_GATT_CHAR_PROP_BIT_READ (*C macro*), 327
ESP_GATT_CHAR_PROP_BIT_WRITE (*C macro*), 327
ESP_GATT_CHAR_PROP_BIT_WRITE_NR (*C macro*), 327
esp_gatt_char_prop_t (*C++ type*), 327
esp_gatt_conn_params_t (*C++ struct*), 319
esp_gatt_conn_params_t::interval (*C++ member*), 320
esp_gatt_conn_params_t::latency (*C++ member*), 320
esp_gatt_conn_params_t::timeout (*C++ member*), 320
esp_gatt_conn_reason_t (*C++ enum*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_CONN_CANCEL (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_FAIL_ESTABL (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_L2C_FAILURE (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_LMP_TIMEOUT (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_NONE (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_TERMINATE_L (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_TERMINATE_P (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_TIMEOUT (*C++ enumerator*), 330
esp_gatt_conn_reason_t::ESP_GATT_CONN_UNKNOWN (*C++ enumerator*), 330
esp_gatt_db_attr_type_t (*C++ enum*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_ALL (*C++ enumerator*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_CHARACTERIST (*C++ enumerator*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_DESCRIPTOR (*C++ enumerator*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_INCLUDED_SER (*C++ enumerator*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_PRIMARY_SERV (*C++ enumerator*), 331
esp_gatt_db_attr_type_t::ESP_GATT_DB_SECONDARY_SE (*C++ enumerator*), 331
ESP_GATT_HEART_RATE_CNTL_POINT (*C macro*), 326

ESP_GATT_HEART_RATE_MEAS (*C macro*), 325
esp_gatt_id_t (*C++ struct*), 317
esp_gatt_id_t::inst_id (*C++ member*), 317
esp_gatt_id_t::uuid (*C++ member*), 317
ESP_GATT_IF_NONE (*C macro*), 327
esp_gatt_if_t (*C++ type*), 327
ESP_GATT_ILLEGAL_HANDLE (*C macro*), 326
ESP_GATT_ILLEGAL_UUID (*C macro*), 326
ESP_GATT_MAX_ATTR_LEN (*C macro*), 327
ESP_GATT_MAX_READ_MULTI_HANDLES (*C macro*), 326
ESP_GATT_PERM_ENCRYPT_KEY_SIZE (*C macro*), 327
ESP_GATT_PERM_READ (*C macro*), 326
ESP_GATT_PERM_READ_AUTHORIZATION (*C macro*), 327
ESP_GATT_PERM_READ_ENC_MITM (*C macro*), 326
ESP_GATT_PERM_READ_ENCRYPTED (*C macro*), 326
esp_gatt_perm_t (*C++ type*), 327
ESP_GATT_PERM_WRITE (*C macro*), 326
ESP_GATT_PERM_WRITE_AUTHORIZATION (*C macro*), 327
ESP_GATT_PERM_WRITE_ENC_MITM (*C macro*), 326
ESP_GATT_PERM_WRITE_ENCRYPTED (*C macro*), 326
ESP_GATT_PERM_WRITE_SIGNED (*C macro*), 327
ESP_GATT_PERM_WRITE_SIGNED_MITM (*C macro*), 327
ESP_GATT_PREP_WRITE_CANCEL (*C macro*), 345
ESP_GATT_PREP_WRITE_EXEC (*C macro*), 345
esp_gatt_prep_write_type (*C++ enum*), 328
esp_gatt_prep_write_type::ESP_GATT_PREP_WRITE_CANCEL (*C++ enumerator*), 328
esp_gatt_prep_write_type::ESP_GATT_PREP_WRITE_EXEC (*C++ enumerator*), 328
ESP_GATT_RSP_BY_APP (*C macro*), 327
esp_gatt_rsp_t (*C++ union*), 316
esp_gatt_rsp_t::attr_value (*C++ member*), 317
esp_gatt_rsp_t::handle (*C++ member*), 317
esp_gatt_srvc_id_t (*C++ struct*), 317
esp_gatt_srvc_id_t::id (*C++ member*), 317
esp_gatt_srvc_id_t::is_primary (*C++ member*), 317
esp_gatt_status_t (*C++ enum*), 328
esp_gatt_status_t::ESP_GATT_ALREADY_OPEN (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_APP_RSP (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_AUTH_FAIL (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_BUSY (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_CANCEL (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_CCC_CFG_ERR (*C++ enumerator*), 330
esp_gatt_status_t::ESP_GATT_CMD_STARTED (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_CONGESTED (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_DB_FULL (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_DUP_REG (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_ENCRYPTED_MITM (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_ENCRYPTED_NO_MITM (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_ERR_UNLIKELY (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_ERROR (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_ILLEGAL_PARAMETER (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_INSUF_AUTHENTICATION (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INSUF_AUTHORIZATION (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INSUF_ENCRYPTION (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INSUF_KEY_SIZE (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INSUF_RESOURCE (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_INTERNAL_ERROR (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_INVALID_ATTR_LEN (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INVALID_CFG (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_INVALID_HANDLE (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INVALID_OFFSET (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_INVALID_PDU (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_MORE (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_NO_RESOURCES (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_NOT_ENCRYPTED (*C++ enumerator*), 329
esp_gatt_status_t::ESP_GATT_NOT_FOUND (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_NOT_LONG (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_OK (*C++ enumerator*), 328
esp_gatt_status_t::ESP_GATT_OUT_OF_RANGE (*C++ enumerator*), 330
esp_gatt_status_t::ESP_GATT_PENDING (*C++ enumerator*), 329

esp_gatt_status_t::ESP_GATT_PRC_IN_PROGRESS (C++ enumerator), 330
esp_gatt_status_t::ESP_GATT_PREPARE_Q_FULL (C++ enumerator), 328
esp_gatt_status_t::ESP_GATT_READ_NOT_PERMIT (C++ enumerator), 328
esp_gatt_status_t::ESP_GATT_REQ_NOT_SUPPORTED (C++ enumerator), 328
esp_gatt_status_t::ESP_GATT_SERVICE_STARTED (C++ enumerator), 329
esp_gatt_status_t::ESP_GATT_STACK_RSP (C++ enumerator), 329
esp_gatt_status_t::ESP_GATT_UNKNOWN_ERROR (C++ enumerator), 330
esp_gatt_status_t::ESP_GATT_UNSUPPORT_GRP_TYPE (C++ enumerator), 328
esp_gatt_status_t::ESP_GATT_WRITE_NOT_PERMIT (C++ enumerator), 328
esp_gatt_status_t::ESP_GATT_WRONG_STATE (C++ enumerator), 329
ESP_GATT_UUID_ALERT_LEVEL (C macro), 324
ESP_GATT_UUID_ALERT_NTF_SVC (C macro), 322
ESP_GATT_UUID_ALERT_STATUS (C macro), 324
ESP_GATT_UUID_Automation_IO_SVC (C macro), 323
ESP_GATT_UUID_BATTERY_LEVEL (C macro), 326
ESP_GATT_UUID_BATTERY_SERVICE_SVC (C macro), 322
ESP_GATT_UUID_BLOOD_PRESSURE_SVC (C macro), 322
ESP_GATT_UUID_BODY_COMPOSITION (C macro), 323
ESP_GATT_UUID_BOND_MANAGEMENT_SVC (C macro), 323
ESP_GATT_UUID_CHAR_AGG_FORMAT (C macro), 323
ESP_GATT_UUID_CHAR_CLIENT_CONFIG (C macro), 323
ESP_GATT_UUID_CHAR_DECLARE (C macro), 323
ESP_GATT_UUID_CHAR_DESCRIPTION (C macro), 323
ESP_GATT_UUID_CHAR_EXT_PROP (C macro), 323
ESP_GATT_UUID_CHAR_PRESENT_FORMAT (C macro), 323
ESP_GATT_UUID_CHAR_SRVR_CONFIG (C macro), 323
ESP_GATT_UUID_CHAR_VALID_RANGE (C macro), 323
ESP_GATT_UUID_CONT_GLUCOSE_MONITOR_SVC (C macro), 323
ESP_GATT_UUID_CSC_FEATURE (C macro), 326
ESP_GATT_UUID_CSC_MEASUREMENT (C macro), 326
ESP_GATT_UUID_CURRENT_TIME (C macro), 324
ESP_GATT_UUID_CURRENT_TIME_SVC (C macro), 322
ESP_GATT_UUID_CYCLING_POWER_SVC (C macro), 323
ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC (C macro), 323
ESP_GATT_UUID_DEVICE_INFO_SVC (C macro), 322
ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR (C macro), 324
ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR (C macro), 324
ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR (C macro), 324
ESP_GATT_UUID_ENVIRONMENTAL_SENSING_SVC (C macro), 323
ESP_GATT_UUID_EXT_RPT_REF_DESCR (C macro), 323
ESP_GATT_UUID_FW_VERSION_STR (C macro), 325
ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL (C macro), 324
ESP_GATT_UUID_GAP_DEVICE_NAME (C macro), 324
ESP_GATT_UUID_GAP_ICON (C macro), 324
ESP_GATT_UUID_GAP_PREF_CONN_PARAM (C macro), 324
ESP_GATT_UUID_GATT_SRV_CHGD (C macro), 324
ESP_GATT_UUID_GLUCOSE_SVC (C macro), 322
ESP_GATT_UUID_GM_CONTEXT (C macro), 325
ESP_GATT_UUID_GM_CONTROL_POINT (C macro), 325
ESP_GATT_UUID_GM_FEATURE (C macro), 325
ESP_GATT_UUID_GM_MEASUREMENT (C macro), 324
ESP_GATT_UUID_HEALTH_THERMOM_SVC (C macro), 322
ESP_GATT_UUID_HEART_RATE_SVC (C macro), 322
ESP_GATT_UUID_HID_BT_KB_INPUT (C macro), 325
ESP_GATT_UUID_HID_BT_KB_OUTPUT (C macro), 325
ESP_GATT_UUID_HID_BT_MOUSE_INPUT (C macro), 325
ESP_GATT_UUID_HID_CONTROL_POINT (C macro), 325
ESP_GATT_UUID_HID_INFORMATION (C macro), 325
ESP_GATT_UUID_HID_PROTO_MODE (C macro), 325
ESP_GATT_UUID_HID_REPORT (C macro), 325
ESP_GATT_UUID_HID_REPORT_MAP (C macro), 325
ESP_GATT_UUID_HID_SVC (C macro), 322
ESP_GATT_UUID_HW_VERSION_STR (C macro), 325
ESP_GATT_UUID_IEEE_DATA (C macro), 325

ESP_GATT_UUID_IMMEDIATE_ALERT_SVC (*C macro*), 322
ESP_GATT_UUID_INCLUDE_SERVICE (*C macro*), 323
ESP_GATT_UUID_LINK_LOSS_SVC (*C macro*), 322
ESP_GATT_UUID_LOCAL_TIME_INFO (*C macro*), 324
ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC (*C macro*), 323
ESP_GATT_UUID_MANU_NAME (*C macro*), 325
ESP_GATT_UUID_MODEL_NUMBER_STR (*C macro*), 325
ESP_GATT_UUID_NEXT_DST_CHANGE_SVC (*C macro*), 322
ESP_GATT_UUID_NUM_DIGITALS_DESCR (*C macro*), 324
ESP_GATT_UUID_NW_STATUS (*C macro*), 324
ESP_GATT_UUID_NW_TRIGGER (*C macro*), 324
ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC (*C macro*), 322
ESP_GATT_UUID_PNP_ID (*C macro*), 325
ESP_GATT_UUID_PRI_SERVICE (*C macro*), 323
ESP_GATT_UUID_REF_TIME_INFO (*C macro*), 324
ESP_GATT_UUID_REF_TIME_UPDATE_SVC (*C macro*), 322
ESP_GATT_UUID_RINGER_CP (*C macro*), 324
ESP_GATT_UUID_RINGER_SETTING (*C macro*), 324
ESP_GATT_UUID_RPT_REF_DESCR (*C macro*), 324
ESP_GATT_UUID_RSC_FEATURE (*C macro*), 326
ESP_GATT_UUID_RSC_MEASUREMENT (*C macro*), 326
ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC (*C macro*), 323
ESP_GATT_UUID_SC_CONTROL_POINT (*C macro*), 326
ESP_GATT_UUID_SCAN_INT_WINDOW (*C macro*), 326
ESP_GATT_UUID_SCAN_PARAMETERS_SVC (*C macro*), 322
ESP_GATT_UUID_SCAN_REFRESH (*C macro*), 326
ESP_GATT_UUID_SEC_SERVICE (*C macro*), 323
ESP_GATT_UUID_SENSOR_LOCATION (*C macro*), 326
ESP_GATT_UUID_SERIAL_NUMBER_STR (*C macro*), 325
ESP_GATT_UUID_SW_VERSION_STR (*C macro*), 325
ESP_GATT_UUID_SYSTEM_ID (*C macro*), 325
ESP_GATT_UUID_TIME_TRIGGER_DESCR (*C macro*), 324
ESP_GATT_UUID_TX_POWER_LEVEL (*C macro*), 324
ESP_GATT_UUID_TX_POWER_SVC (*C macro*), 322
ESP_GATT_UUID_USER_DATA_SVC (*C macro*), 323
ESP_GATT_UUID_VALUE_TRIGGER_DESCR (*C macro*), 324
ESP_GATT_UUID_WEIGHT_SCALE_SVC (*C macro*), 323
esp_gatt_value_t (*C++ struct*), 319
esp_gatt_value_t::auth_req (*C++ member*), 319
esp_gatt_value_t::handle (*C++ member*), 319
esp_gatt_value_t::len (*C++ member*), 319
esp_gatt_value_t::offset (*C++ member*), 319
esp_gatt_value_t::value (*C++ member*), 319
esp_gatt_write_type_t (*C++ enum*), 331
esp_gatt_write_type_t::ESP_GATT_WRITE_TYPE_NO_RSP (*C++ enumerator*), 331
esp_gatt_write_type_t::ESP_GATT_WRITE_TYPE_RSP (*C++ enumerator*), 331
esp_gattc_cb_event_t (*C++ enum*), 363
esp_gattc_cb_event_t::ESP_GATTC_ACL_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_ADV_DATA_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_ADV_VSC_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_CFG_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_DIS_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_ENB_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_PARAM_EV (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_RD_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_BTH_SCAN_THR_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_CANCEL_OPEN_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_CFG_MTU_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_CLOSE_EVT (*C++ enumerator*), 363
esp_gattc_cb_event_t::ESP_GATTC_CONGEST_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_CONNECT_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_DIS_SRVC_CMPL_EVT (*C++ enumerator*), 366
esp_gattc_cb_event_t::ESP_GATTC_DISCONNECT_EVT (*C++ enumerator*), 365
esp_gattc_cb_event_t::ESP_GATTC_ENC_CMPL_CB_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_EXEC_EVT (*C++ enumerator*), 364
esp_gattc_cb_event_t::ESP_GATTC_GET_ADDR_LIST_EVT (*C++ enumerator*), 366

esp_gattc_cb_event_t::ESP_GATTC_MULT_ADV_DATA_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_MULT_ADV_DISB_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_MULT_ADV_ENB_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_MULT_ADV_UPD_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_NOTIFY_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_OPEN_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_PREP_WRITE_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_QUEUE_FULL_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_READ_CHAR_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_READ_DESCR_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_READ_MULTI_VAR_EVT (C++ enumerator), 366
esp_gattc_cb_event_t::ESP_GATTC_READ_MULTIPLE_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_REG_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_REG_FOR_NOTIFY_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_SCAN_FLT_CFG_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_SCAN_FLT_PARAM_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_SCAN_FLT_STATUS_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_SEARCH_CMPL_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_SEARCH_RES_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_SET_ASSOC_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_SRVC_CHG_EVT (C++ enumerator), 364
esp_gattc_cb_event_t::ESP_GATTC_UNREG_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_UNREG_FOR_NOTIFY_EVT (C++ enumerator), 365
esp_gattc_cb_event_t::ESP_GATTC_WRITE_CHAR_EVT (C++ enumerator), 363
esp_gattc_cb_event_t::ESP_GATTC_WRITE_DESCR_EVT (C++ enumerator), 364
esp_gattc_cb_t (C++ type), 363
esp_gattc_char_elem_t (C++ struct), 321
esp_gattc_char_elem_t::char_handle (C++ member), 321
esp_gattc_char_elem_t::properties (C++ member), 321
esp_gattc_char_elem_t::uuid (C++ member), 321
esp_gattc_db_elem_t (C++ struct), 320
esp_gattc_db_elem_t::attribute_handle (C++ member), 320
esp_gattc_db_elem_t::end_handle (C++ member), 320
esp_gattc_db_elem_t::properties (C++ member), 320
esp_gattc_db_elem_t::start_handle (C++ member), 320
esp_gattc_db_elem_t::type (C++ member), 320
esp_gattc_db_elem_t::uuid (C++ member), 320
esp_gattc_descr_elem_t (C++ struct), 321
esp_gattc_descr_elem_t::handle (C++ member), 321
esp_gattc_descr_elem_t::uuid (C++ member), 321
esp_gattc_incl_svc_elem_t (C++ struct), 321
esp_gattc_incl_svc_elem_t::handle (C++ member), 322
esp_gattc_incl_svc_elem_t::incl_srvc_e_handle (C++ member), 322
esp_gattc_incl_svc_elem_t::incl_srvc_s_handle (C++ member), 322
esp_gattc_incl_svc_elem_t::uuid (C++ member), 322
esp_gattc_multi_t (C++ struct), 320
esp_gattc_multi_t::handles (C++ member), 320
esp_gattc_multi_t::num_attr (C++ member), 320
esp_gattc_service_elem_t (C++ struct), 320
esp_gattc_service_elem_t::end_handle (C++ member), 321
esp_gattc_service_elem_t::is_primary (C++ member), 321
esp_gattc_service_elem_t::start_handle (C++ member), 321
esp_gattc_service_elem_t::uuid (C++ member), 321
esp_gatts_attr_db_t (C++ struct), 318
esp_gatts_attr_db_t::att_desc (C++ member), 318
esp_gatts_attr_db_t::attr_control (C++ member), 318
esp_gatts_cb_event_t (C++ enum), 345
esp_gatts_cb_event_t::ESP_GATTS_ADD_CHAR_DESCR_EVT (C++ enumerator), 346
esp_gatts_cb_event_t::ESP_GATTS_ADD_CHAR_EVT (C++ enumerator), 346
esp_gatts_cb_event_t::ESP_GATTS_ADD_INCL_SRVC_EVT (C++ enumerator), 346
esp_gatts_cb_event_t::ESP_GATTS_CANCEL_OPEN_EVT (C++ enumerator), 346
esp_gatts_cb_event_t::ESP_GATTS_CLOSE_EVT (C++ enumerator), 346
esp_gatts_cb_event_t::ESP_GATTS_CONF_EVT

(C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_CONGEST_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_CONNECT_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_CREAT_ATTR_TAB_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_CREATE_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_DELETE_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_DISCONNECT_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_EXEC_WRITE_EVT (C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_LISTEN_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_MTU_EVT (C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_OPEN_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_READ_EVT (C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_REG_EVT (C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_RESPONSE_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_SEND_SERVICE_CHANGE_EVT (C++ *enumerator*), 347
esp_gatts_cb_event_t::ESP_GATTS_SET_ATTR_VAL_EVT (C++ *enumerator*), 347
esp_gatts_cb_event_t::ESP_GATTS_START_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_STOP_EVT (C++ *enumerator*), 346
esp_gatts_cb_event_t::ESP_GATTS_UNREG_EVT (C++ *enumerator*), 345
esp_gatts_cb_event_t::ESP_GATTS_WRITE_EVT (C++ *enumerator*), 345
esp_gatts_cb_t (C++ *type*), 345
esp_gatts_incl128_svc_desc_t (C++ *struct*), 319
esp_gatts_incl128_svc_desc_t::end_hdl (C++ *member*), 319
esp_gatts_incl128_svc_desc_t::start_hdl (C++ *member*), 319
esp_gatts_incl_svc_desc_t (C++ *struct*), 318
esp_gatts_incl_svc_desc_t::end_hdl (C++ *member*), 319
esp_gatts_incl_svc_desc_t::start_hdl (C++ *member*), 319
esp_gatts_incl_svc_desc_t::uuid (C++ *member*), 319
esp_gcov_dump (C++ *function*), 1835
esp_get_deep_sleep_wake_stub (C++ *function*), 2131
esp_get_flash_encryption_mode (C++ *function*), 1802
esp_get_free_heap_size (C++ *function*), 2084
esp_get_free_internal_heap_size (C++ *function*), 2084
esp_get_idf_version (C++ *function*), 2086
esp_get_minimum_free_heap_size (C++ *function*), 2084
ESP_GOTO_ON_ERROR (C *macro*), 1870
ESP_GOTO_ON_ERROR_ISR (C *macro*), 1870
ESP_GOTO_ON_FALSE (C *macro*), 1870
ESP_GOTO_ON_FALSE_ISR (C *macro*), 1870
esp_hd_cb_t (C++ *type*), 490
esp_hf_ag_answer_call (C++ *function*), 471
esp_hf_ag_audio_connect (C++ *function*), 468
esp_hf_ag_audio_disconnect (C++ *function*), 468
esp_hf_ag_bsir (C++ *function*), 471
esp_hf_ag_ciev_report (C++ *function*), 469
esp_hf_ag_cind_response (C++ *function*), 470
esp_hf_ag_clcc_response (C++ *function*), 470
esp_hf_ag_cmee_send (C++ *function*), 469
esp_hf_ag_cnum_response (C++ *function*), 471
esp_hf_ag_cops_response (C++ *function*), 470
esp_hf_ag_deinit (C++ *function*), 467
esp_hf_ag_devices_status_indchange (C++ *function*), 469
esp_hf_ag_end_call (C++ *function*), 472
esp_hf_ag_init (C++ *function*), 467
esp_hf_ag_out_call (C++ *function*), 472
esp_hf_ag_outgoing_data_ready (C++ *function*), 473
esp_hf_ag_pkt_stat_nums_get (C++ *function*), 473
esp_hf_ag_register_callback (C++ *function*), 467
esp_hf_ag_register_data_callback (C++ *function*), 472
esp_hf_ag_reject_call (C++ *function*), 471
esp_hf_ag_slc_connect (C++ *function*), 468
esp_hf_ag_slc_disconnect (C++ *function*), 468
esp_hf_ag_unknown_at_send (C++ *function*), 469
esp_hf_ag_volume_control (C++ *function*), 468
esp_hf_ag_vra_control (C++ *function*), 468
esp_hf_at_response_code_t (C++ *enum*), 449
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_COD (C++ *enumerator*), 450

esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_CODE_ERR (C++ enumerator), 445
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_CODE_OK (C++ enumerator), 450
esp_hf_at_response_code_t::ESP_HF_AT_RESPONSE_CODE_OK (C++ enumerator), 449
esp_hf_at_response_t (C++ enum), 450
esp_hf_at_response_t::ESP_HF_AT_RESPONSE_ERROR (C++ enumerator), 450
esp_hf_at_response_t::ESP_HF_AT_RESPONSE_OK (C++ enumerator), 450
esp_hf_audio_state_t (C++ enum), 443
esp_hf_audio_state_t::ESP_HF_AUDIO_STATE_CONNECTED (C++ enumerator), 444
esp_hf_audio_state_t::ESP_HF_AUDIO_STATE_CONNECTED_MSBC (C++ enumerator), 444
esp_hf_audio_state_t::ESP_HF_AUDIO_STATE_CONNECTING (C++ enumerator), 444
esp_hf_audio_state_t::ESP_HF_AUDIO_STATE_DISCONNECTED (C++ enumerator), 443
esp_hf_btrh_cmd_t (C++ enum), 448
esp_hf_btrh_cmd_t::ESP_HF_BTRH_CMD_ACCEPT (C++ enumerator), 448
esp_hf_btrh_cmd_t::ESP_HF_BTRH_CMD_HOLD (C++ enumerator), 448
esp_hf_btrh_cmd_t::ESP_HF_BTRH_CMD_REJECT (C++ enumerator), 448
esp_hf_btrh_status_t (C++ enum), 447
esp_hf_btrh_status_t::ESP_HF_BTRH_STATUS_ACCEPTED (C++ enumerator), 448
esp_hf_btrh_status_t::ESP_HF_BTRH_STATUS_HELD (C++ enumerator), 447
esp_hf_btrh_status_t::ESP_HF_BTRH_STATUS_REJECTED (C++ enumerator), 448
esp_hf_call_addr_type_t (C++ enum), 447
esp_hf_call_addr_type_t::ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL (C++ enumerator), 447
esp_hf_call_addr_type_t::ESP_HF_CALL_ADDR_TYPE_UNKNOWN (C++ enumerator), 447
esp_hf_call_held_status_t (C++ enum), 445
esp_hf_call_held_status_t::ESP_HF_CALL_HELD_STATUS_HELD (C++ enumerator), 446
esp_hf_call_held_status_t::ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE (C++ enumerator), 446
esp_hf_call_held_status_t::ESP_HF_CALL_HELD_STATUS_NONE (C++ enumerator), 446
esp_hf_call_setup_status_t (C++ enum), 445
esp_hf_call_setup_status_t::ESP_HF_CALL_SETUP_STATUS_IDLE (C++ enumerator), 445
esp_hf_call_setup_status_t::ESP_HF_CALL_SETUP_STATUS_INCOMING (C++ enumerator), 445
esp_hf_call_setup_status_t::ESP_HF_CALL_SETUP_STATUS_INCOMING_ALERTING (C++ enumerator), 445
esp_hf_call_setup_status_t::ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING (C++ enumerator), 445
esp_hf_call_status_t (C++ enum), 445
esp_hf_call_status_t::ESP_HF_CALL_STATUS_CALL_IN_PROGRESS (C++ enumerator), 445
esp_hf_call_status_t::ESP_HF_CALL_STATUS_NO_CALLS (C++ enumerator), 445
esp_hf_call_waiting_status_t (C++ enum), 448
esp_hf_call_waiting_status_t::ESP_HF_CALL_WAITING_STATUS_ACTIVE (C++ enumerator), 448
esp_hf_call_waiting_status_t::ESP_HF_CALL_WAITING_STATUS_INACTIVE (C++ enumerator), 448
esp_hf_cb_event_t (C++ enum), 480
esp_hf_cb_event_t::ESP_HF_ATA_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_AUDIO_STATE_EVT (C++ enumerator), 480
esp_hf_cb_event_t::ESP_HF_BCS_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_BVRA_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_CHUP_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_CIND_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_CLCC_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_CNUM_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_CONNECTION_STATE_EVT (C++ enumerator), 480
esp_hf_cb_event_t::ESP_HF_COPS_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_DIAL_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_IND_UPDATE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_NREC_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_PKT_STAT_NUMS_GET_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_UNAT_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_VOLUME_CONTROL_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_VTS_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_event_t::ESP_HF_WBS_RESPONSE_EVT (C++ enumerator), 481
esp_hf_cb_param_t (C++ union), 473
esp_hf_cb_param_t::ag_pkt_status_nums (C++ struct), 474
esp_hf_cb_param_t::ag_pkt_status_nums::rx_correct (C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::rx_err (C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::rx_lost (C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::rx_none (C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::rx_total (C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::tx_discard

(C++ member), 475
esp_hf_cb_param_t::ag_pkt_status_nums::esp_hf_cb_param_t::hf_conn_stat_param::peer_feat (C++ member), 475
esp_hf_cb_param_t::ata_rep (C++ member), 474
esp_hf_cb_param_t::audio_stat (C++ member), 473
esp_hf_cb_param_t::bcs_rep (C++ member), 474
esp_hf_cb_param_t::chup_rep (C++ member), 474
esp_hf_cb_param_t::cind_rep (C++ member), 474
esp_hf_cb_param_t::clcc_rep (C++ member), 474
esp_hf_cb_param_t::cnum_rep (C++ member), 474
esp_hf_cb_param_t::conn_stat (C++ member), 473
esp_hf_cb_param_t::cops_rep (C++ member), 474
esp_hf_cb_param_t::hf_ata_rep_param (C++ struct), 475
esp_hf_cb_param_t::hf_ata_rep_param::remote_bda (C++ member), 475
esp_hf_cb_param_t::hf_audio_stat_param (C++ struct), 475
esp_hf_cb_param_t::hf_audio_stat_param::esp_hf_cb_param_t::hf_conn_stat_param::remote_bda (C++ member), 475
esp_hf_cb_param_t::hf_audio_stat_param::state (C++ member), 475
esp_hf_cb_param_t::hf_audio_stat_param::sync_conn_handle (C++ member), 475
esp_hf_cb_param_t::hf_bcs_rep_param (C++ struct), 475
esp_hf_cb_param_t::hf_bcs_rep_param::mode (C++ member), 476
esp_hf_cb_param_t::hf_bcs_rep_param::remote_bda (C++ member), 476
esp_hf_cb_param_t::hf_chup_rep_param (C++ struct), 476
esp_hf_cb_param_t::hf_chup_rep_param::remote_bda (C++ member), 476
esp_hf_cb_param_t::hf_cind_rep_param (C++ struct), 476
esp_hf_cb_param_t::hf_cind_rep_param::remote_bda (C++ member), 476
esp_hf_cb_param_t::hf_clcc_rep_param (C++ struct), 476
esp_hf_cb_param_t::hf_clcc_rep_param::remote_bda (C++ member), 476
esp_hf_cb_param_t::hf_cnum_rep_param (C++ struct), 476
esp_hf_cb_param_t::hf_cnum_rep_param::remote_bda (C++ member), 476
esp_hf_cb_param_t::hf_conn_stat_param (C++ struct), 476
esp_hf_cb_param_t::hf_conn_stat_param::chld_feat (C++ member), 477
esp_hf_cb_param_t::hf_conn_stat_param::remote_bda (C++ member), 477
esp_hf_cb_param_t::hf_conn_stat_param::state (C++ member), 477
esp_hf_cb_param_t::hf_cops_rep_param (C++ struct), 477
esp_hf_cb_param_t::hf_cops_rep_param::remote_addr (C++ member), 477
esp_hf_cb_param_t::hf_ind_upd_param (C++ struct), 477
esp_hf_cb_param_t::hf_ind_upd_param::remote_addr (C++ member), 477
esp_hf_cb_param_t::hf_nrec_param (C++ struct), 477
esp_hf_cb_param_t::hf_nrec_param::remote_addr (C++ member), 477
esp_hf_cb_param_t::hf_nrec_param::state (C++ member), 477
esp_hf_cb_param_t::hf_out_call_param (C++ struct), 477
esp_hf_cb_param_t::hf_out_call_param::num_or_loc (C++ member), 478
esp_hf_cb_param_t::hf_out_call_param::remote_addr (C++ member), 478
esp_hf_cb_param_t::hf_out_call_param::type (C++ member), 478
esp_hf_cb_param_t::hf_unat_rep_param (C++ struct), 478
esp_hf_cb_param_t::hf_unat_rep_param::remote_addr (C++ member), 478
esp_hf_cb_param_t::hf_unat_rep_param::unat (C++ member), 478
esp_hf_cb_param_t::hf_volume_control_param (C++ struct), 478
esp_hf_cb_param_t::hf_volume_control_param::remote_addr (C++ member), 478
esp_hf_cb_param_t::hf_volume_control_param::type (C++ member), 478
esp_hf_cb_param_t::hf_volume_control_param::volume (C++ member), 478
esp_hf_cb_param_t::hf_vra_rep_param (C++ struct), 478
esp_hf_cb_param_t::hf_vra_rep_param::remote_addr (C++ member), 478
esp_hf_cb_param_t::hf_vra_rep_param::value (C++ member), 478
esp_hf_cb_param_t::hf_vts_rep_param (C++ struct), 478
esp_hf_cb_param_t::hf_vts_rep_param::code (C++ member), 479
esp_hf_cb_param_t::hf_vts_rep_param::remote_addr (C++ member), 479
esp_hf_cb_param_t::hf_wbs_rep_param (C++ struct), 479
esp_hf_cb_param_t::hf_wbs_rep_param::codec

(C++ *member*), 479
esp_hf_cb_param_t::hf_wbs_rep_param::reserved (C++ *member*), 479
esp_hf_cb_param_t::ind_upd (C++ *member*), 474
esp_hf_cb_param_t::nrec (C++ *member*), 474
esp_hf_cb_param_t::out_call (C++ *member*), 474
esp_hf_cb_param_t::pkt_nums (C++ *member*), 474
esp_hf_cb_param_t::unat_rep (C++ *member*), 474
esp_hf_cb_param_t::volume_control (C++ *member*), 473
esp_hf_cb_param_t::vra_rep (C++ *member*), 473
esp_hf_cb_param_t::vts_rep (C++ *member*), 474
esp_hf_cb_param_t::wbs_rep (C++ *member*), 474
esp_hf_cb_t (C++ *type*), 480
ESP_HF_CHLD_FEAT_HOLD_ACC (C *macro*), 480
ESP_HF_CHLD_FEAT_MERGE (C *macro*), 480
ESP_HF_CHLD_FEAT_MERGE_DETACH (C *macro*), 480
ESP_HF_CHLD_FEAT_PRIV_X (C *macro*), 480
ESP_HF_CHLD_FEAT_REL (C *macro*), 480
ESP_HF_CHLD_FEAT_REL_ACC (C *macro*), 480
ESP_HF_CHLD_FEAT_REL_X (C *macro*), 480
esp_hf_chld_type_t (C++ *enum*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_HOLD_ACC (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_MERGE (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_MERGE_DETACH (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_PRIV_X (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_REL (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_REL_ACC (C++ *enumerator*), 449
esp_hf_chld_type_t::ESP_HF_CHLD_TYPE_REL_X (C++ *enumerator*), 449
esp_hf_ciev_report_type_t (C++ *enum*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_BATTCHG (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_CALL (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_CALLHELD (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_CALLSETUP (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_ROAM (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_SERVICE (C++ *enumerator*), 444
esp_hf_ciev_report_type_t::ESP_HF_IND_TYPE_SIGNAL (C++ *enumerator*), 444
esp_hf_client_answer_call (C++ *function*), 454
esp_hf_client_audio_state_t (C++ *enum*), 465
esp_hf_client_audio_state_t::ESP_HF_CLIENT_AUDIO_ (C++ *enumerator*), 465
esp_hf_client_audio_state_t::ESP_HF_CLIENT_AUDIO_ (C++ *enumerator*), 465
esp_hf_client_audio_state_t::ESP_HF_CLIENT_AUDIO_ (C++ *enumerator*), 465
esp_hf_client_audio_state_t::ESP_HF_CLIENT_AUDIO_ (C++ *enumerator*), 465
esp_hf_client_cb_event_t (C++ *enum*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_AT_RESPON (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_AUDIO_STA (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_BINP_EVT (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_BSIR_EVT (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_BTRH_EVT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_BVRA_EVT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CCWA_EVT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_BATT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_CALL (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_CALL (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_CALL (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_ROAM (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_SERV (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CIND_SIGN (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CLCC_EVT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CLIP_EVT (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CNUM_EVT (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_CONNECTIO (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_COPS_CURR (C++ *enumerator*), 466
esp_hf_client_cb_event_t::ESP_HF_CLIENT_PKT_STAT_ (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_RING_IND_ (C++ *enumerator*), 467
esp_hf_client_cb_event_t::ESP_HF_CLIENT_VOLUME_CO (C++ *enumerator*), 467

esp_hf_client_cb_param_t (*C++ union*), 457
esp_hf_client_cb_param_t::at_response (*C++ member*), 458
esp_hf_client_cb_param_t::audio_stat (*C++ member*), 457
esp_hf_client_cb_param_t::battery_level (*C++ member*), 457
esp_hf_client_cb_param_t::binp (*C++ member*), 458
esp_hf_client_cb_param_t::bsir (*C++ member*), 458
esp_hf_client_cb_param_t::btrh (*C++ member*), 457
esp_hf_client_cb_param_t::bvra (*C++ member*), 457
esp_hf_client_cb_param_t::call (*C++ member*), 457
esp_hf_client_cb_param_t::call_held (*C++ member*), 457
esp_hf_client_cb_param_t::call_setup (*C++ member*), 457
esp_hf_client_cb_param_t::ccwa (*C++ member*), 458
esp_hf_client_cb_param_t::clcc (*C++ member*), 458
esp_hf_client_cb_param_t::clip (*C++ member*), 457
esp_hf_client_cb_param_t::cnum (*C++ member*), 458
esp_hf_client_cb_param_t::conn_stat (*C++ member*), 457
esp_hf_client_cb_param_t::cops (*C++ member*), 457
esp_hf_client_cb_param_t::hf_client_at_response (*C++ struct*), 458
esp_hf_client_cb_param_t::hf_client_at_response_param::cme (*C++ member*), 458
esp_hf_client_cb_param_t::hf_client_at_response_param::code (*C++ member*), 458
esp_hf_client_cb_param_t::hf_client_audio_stat (*C++ struct*), 458
esp_hf_client_cb_param_t::hf_client_audio_stat_param::data (*C++ member*), 458
esp_hf_client_cb_param_t::hf_client_audio_stat_param::state (*C++ member*), 458
esp_hf_client_cb_param_t::hf_client_audio_stat_param::sync_conn_handle (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_battery_level (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_battery_level_param::value (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_binp (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_binp_param::number (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_bsir (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_bsir_param::state (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_btrh (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_btrh_param::status (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_bvra (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_bvra_param::value (*C++ member*), 459
esp_hf_client_cb_param_t::hf_client_call_held_ind (*C++ struct*), 459
esp_hf_client_cb_param_t::hf_client_call_held_ind_param::status (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_call_ind_param (*C++ struct*), 460
esp_hf_client_cb_param_t::hf_client_call_ind_param::status (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_call_setup_ind (*C++ struct*), 460
esp_hf_client_cb_param_t::hf_client_call_setup_ind_param::status (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_ccwa_param (*C++ struct*), 460
esp_hf_client_cb_param_t::hf_client_ccwa_param::number (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param (*C++ struct*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param::dir (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param::idx (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param::mode (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param::mpty (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_clcc_param::number (*C++ member*), 460
esp_hf_client_cb_param_t::hf_client_clcc_param::status (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_clip_param (*C++ struct*), 461
esp_hf_client_cb_param_t::hf_client_clip_param::number (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_cnum_param (*C++ struct*), 461
esp_hf_client_cb_param_t::hf_client_cnum_param::number (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_cnum_param::type (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param (*C++ struct*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param::call_setup_ind (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param::call_setup_ind_value (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param::call_state (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param::call_state_ind (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_conn_stat_param::service_availability_status (*C++ member*), 461
esp_hf_client_cb_param_t::hf_client_current_operator (*C++ member*), 461

(C++ *struct*), 461
esp_hf_client_cb_param_t::hf_client_current_operator (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_network_roaming (C++ *struct*), 462
esp_hf_client_cb_param_t::hf_client_network_roaming::status (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat (C++ *struct*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::nb_rec (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::nb_sent (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::tx_lost (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::tx_nerr (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::tx_total (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_pkt_stat::wide_band (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_service_availability (C++ *struct*), 462
esp_hf_client_cb_param_t::hf_client_service_availability::status (C++ *member*), 462
esp_hf_client_cb_param_t::hf_client_signal_strength_ind (C++ *struct*), 462
esp_hf_client_cb_param_t::hf_client_signal_strength_ind::value (C++ *member*), 463
esp_hf_client_cb_param_t::hf_client_volume_control (C++ *struct*), 463
esp_hf_client_cb_param_t::hf_client_volume_control::type (C++ *member*), 463
esp_hf_client_cb_param_t::hf_client_volume_control::volume (C++ *member*), 463
esp_hf_client_cb_param_t::pkt_nums (C++ *member*), 458
esp_hf_client_cb_param_t::roaming (C++ *member*), 457
esp_hf_client_cb_param_t::service_availability (C++ *member*), 457
esp_hf_client_cb_param_t::signal_strength (C++ *member*), 457
esp_hf_client_cb_param_t::volume_control (C++ *member*), 458
esp_hf_client_cb_t (C++ *type*), 465
ESP_HF_CLIENT_CHLD_FEAT_HOLD_ACC (C *macro*), 464
ESP_HF_CLIENT_CHLD_FEAT_MERGE (C *macro*), 464
ESP_HF_CLIENT_CHLD_FEAT_MERGE_DETACH (C *macro*), 464
ESP_HF_CLIENT_CHLD_FEAT_PRIV_X (C *macro*), 464
ESP_HF_CLIENT_CHLD_FEAT_REL (C *macro*), 464
ESP_HF_CLIENT_CHLD_FEAT_REL_ACC (C *macro*), 464
esp_hf_client_cb_param_t::name (C++ *member*), 464
ESP_HF_CLIENT_CHLD_FEAT_REL_X (C *macro*), 464
esp_hf_client_connect (C++ *function*), 452
esp_hf_client_connect_audio (C++ *function*), 453
esp_hf_client_connection_state_t (C++ *enum*), 465
esp_hf_client_connection_state_t::ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED (C++ *enumerator*), 465
esp_hf_client_connection_state_t::ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING (C++ *enumerator*), 465
esp_hf_client_connection_state_t::ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED (C++ *enumerator*), 465
esp_hf_client_connection_state_t::ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING (C++ *enumerator*), 465
esp_hf_client_connection_state_t::ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED (C++ *enumerator*), 465
esp_hf_client_deinit (C++ *function*), 452
esp_hf_client_dial (C++ *function*), 453
esp_hf_client_dial_memory (C++ *function*), 454
esp_hf_client_register_data_callback (C++ *function*), 452
esp_hf_client_register_data_callback::incoming_cb (C++ *member*), 452
esp_hf_client_register_data_callback::outgoing_cb (C++ *member*), 452
esp_hf_client_register_state_callback (C++ *function*), 453
esp_hf_client_service_availability_status_t (C++ *enum*), 465
esp_hf_client_service_availability_status_t::ESP_HF_CLIENT_SERVICE_AVAILABILITY_STATUS_NOT_AVAILABLE (C++ *enumerator*), 465
esp_hf_client_service_availability_status_t::ESP_HF_CLIENT_SERVICE_AVAILABILITY_STATUS_UNKNOWN (C++ *enumerator*), 465
esp_hf_client_service_availability_status_t::ESP_HF_CLIENT_SERVICE_AVAILABILITY_STATUS_AVAILABLE (C++ *enumerator*), 465
esp_hf_client_outgoing_data_cb_t (C++ *type*), 464
esp_hf_client_outgoing_data_ready (C++ *function*), 456
esp_hf_client_pcm_resample (C++ *function*), 456
esp_hf_client_pcm_resample_deinit (C++ *function*), 456
esp_hf_client_pcm_resample_init (C++ *function*), 456
ESP_HF_CLIENT_PEER_FEAT_3WAY (C *macro*), 463
ESP_HF_CLIENT_PEER_FEAT_CODEC (C *macro*), 463
ESP_HF_CLIENT_PEER_FEAT_ECC (C *macro*), 463
ESP_HF_CLIENT_PEER_FEAT_ECNR (C *macro*), 463
ESP_HF_CLIENT_PEER_FEAT_ECS (C *macro*), 463
ESP_HF_CLIENT_PEER_FEAT_ESCO_S4 (C *macro*), 464

ESP_HF_CLIENT_PEER_FEAT_EXTERR (*C macro*), 463
ESP_HF_CLIENT_PEER_FEAT_HF_IND (*C macro*), 464
ESP_HF_CLIENT_PEER_FEAT_INBAND (*C macro*), 463
ESP_HF_CLIENT_PEER_FEAT_REJECT (*C macro*), 463
ESP_HF_CLIENT_PEER_FEAT_VREC (*C macro*), 463
ESP_HF_CLIENT_PEER_FEAT_VTAG (*C macro*), 463
esp_hf_client_pkt_stat_nums_get (*C++ function*), 456
esp_hf_client_query_current_calls (*C++ function*), 454
esp_hf_client_query_current_operator_name (*C++ function*), 455
esp_hf_client_register_callback (*C++ function*), 452
esp_hf_client_register_data_callback (*C++ function*), 456
esp_hf_client_reject_call (*C++ function*), 454
esp_hf_client_request_last_voice_tag_number (*C++ function*), 455
esp_hf_client_retrieve_subscriber_info (*C++ function*), 455
esp_hf_client_send_btrh_cmd (*C++ function*), 454
esp_hf_client_send_chld_cmd (*C++ function*), 454
esp_hf_client_send_dtmf (*C++ function*), 455
esp_hf_client_send_iphoneaccev (*C++ function*), 455
esp_hf_client_send_nrec (*C++ function*), 456
esp_hf_client_send_xapl (*C++ function*), 455
esp_hf_client_start_voice_recognition (*C++ function*), 453
esp_hf_client_stop_voice_recognition (*C++ function*), 453
esp_hf_client_volume_update (*C++ function*), 453
ESP_HF_CLIENT_XAPL_FEAT_BATTERY_REPORT (*C macro*), 464
ESP_HF_CLIENT_XAPL_FEAT_DOCKED (*C macro*), 464
ESP_HF_CLIENT_XAPL_FEAT_RESERVED (*C macro*), 464
ESP_HF_CLIENT_XAPL_FEAT_SIRI_STATUS_REPORT (*C macro*), 464
ESP_HF_CLIENT_XAPL_NR_STATUS_REPORT (*C macro*), 464
esp_hf_cme_err_t (*C++ enum*), 450
esp_hf_cme_err_t::ESP_HF_CME_AG_FAILURE (*C++ enumerator*), 450
esp_hf_cme_err_t::ESP_HF_CME_DIAL_STRING_TOO_LONG (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_INCORRECT_PASSWORD (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_INVALID_CHARACTERS_I (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_INVALID_CHARACTERS_I (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_INVALID_INDEX (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_MEMORY_FAILURE (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_MEMORY_FULL (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_NETWORK_NOT_ALLOWED (*C++ enumerator*), 452
esp_hf_cme_err_t::ESP_HF_CME_NETWORK_TIMEOUT (*C++ enumerator*), 452
esp_hf_cme_err_t::ESP_HF_CME_NO_CONNECTION_TO_PHO (*C++ enumerator*), 450
esp_hf_cme_err_t::ESP_HF_CME_NO_NETWORK_SERVICE (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_OPERATION_NOT_ALLOWE (*C++ enumerator*), 450
esp_hf_cme_err_t::ESP_HF_CME_OPERATION_NOT_SUPPOR (*C++ enumerator*), 450
esp_hf_cme_err_t::ESP_HF_CME_PH_SIM_PIN_REQUIRED (*C++ enumerator*), 450
esp_hf_cme_err_t::ESP_HF_CME_SIM_BUSY (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_FAILURE (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_NOT_INSERTED (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_PIN2_REQUIRED (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_PIN_REQUIRED (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_PUK2_REQUIRED (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_SIM_PUK_REQUIRED (*C++ enumerator*), 451
esp_hf_cme_err_t::ESP_HF_CME_TEXT_STRING_TOO_LONG (*C++ enumerator*), 451
esp_hf_connection_state_t (*C++ enum*), 448
esp_hf_connection_state_t::ESP_HF_CONNECTION_STAT (*C++ enumerator*), 449
esp_hf_connection_state_t::ESP_HF_CONNECTION_STAT (*C++ enumerator*), 449
esp_hf_connection_state_t::ESP_HF_CONNECTION_STAT (*C++ enumerator*), 449
esp_hf_connection_state_t::ESP_HF_CONNECTION_STAT (*C++ enumerator*), 449
esp_hf_connection_state_t::ESP_HF_CONNECTION_STAT (*C++ enumerator*), 449
esp_hf_current_call_direction_t (*C++ enum*), 446
esp_hf_current_call_direction_t::ESP_HF_CURRENT_C (*C++ enumerator*), 446
esp_hf_current_call_direction_t::ESP_HF_CURRENT_C

(C++ enumerator), 446
esp_hf_current_call_mode_t (C++ enum), 447
esp_hf_current_call_mode_t::ESP_HF_CURRENT_CALL_MODE_DATA (C++ enumerator), 447
esp_hf_current_call_mode_t::ESP_HF_CURRENT_CALL_MODE_DATA_ESCO_S4 (C++ enumerator), 447
esp_hf_current_call_mode_t::ESP_HF_CURRENT_CALL_MODE_MAX (C++ enumerator), 447
esp_hf_current_call_mpty_type_t (C++ enum), 446
esp_hf_current_call_mpty_type_t::ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI (C++ enumerator), 447
esp_hf_current_call_mpty_type_t::ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE (C++ enumerator), 447
esp_hf_current_call_status_t (C++ enum), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_ACTIVE (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_ALERTING (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_DIALING (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_HELD (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_INCOMING (C++ enumerator), 446
esp_hf_current_call_status_t::ESP_HF_CURRENT_CALL_STATUS_WAITING (C++ enumerator), 446
esp_hf_dial_type_t (C++ enum), 482
esp_hf_dial_type_t::ESP_HF_DIAL_MEM (C++ enumerator), 482
esp_hf_dial_type_t::ESP_HF_DIAL_NUM (C++ enumerator), 482
esp_hf_dial_type_t::ESP_HF_DIAL_VOIP (C++ enumerator), 482
esp_hf_in_band_ring_state_t (C++ enum), 443
esp_hf_in_band_ring_state_t::ESP_HF_IN_BAND_RINGTONE_NOT_PROVIDED (C++ enumerator), 443
esp_hf_in_band_ring_state_t::ESP_HF_IN_BAND_RINGTONE_PROVIDED (C++ enumerator), 443
esp_hf_incoming_data_cb_t (C++ type), 480
esp_hf_network_state_t (C++ enum), 444
esp_hf_network_state_t::ESP_HF_NETWORK_STATE_AVAILABLE (C++ enumerator), 444
esp_hf_network_state_t::ESP_HF_NETWORK_STATE_NOT_AVAILABLE (C++ enumerator), 444
esp_hf_nrec_t (C++ enum), 448
esp_hf_nrec_t::ESP_HF_NREC_START (C++ enumerator), 448
esp_hf_nrec_t::ESP_HF_NREC_STOP (C++ enumerator), 448
esp_hf_outgoing_data_cb_t (C++ type), 480
ESP_HF_PEER_FEAT_3WAY (C macro), 479
ESP_HF_PEER_FEAT_CODEC (C macro), 479
ESP_HF_PEER_FEAT_ECC (C macro), 479
ESP_HF_PEER_FEAT_ECNR (C macro), 479
ESP_HF_PEER_FEAT_ECS (C macro), 479
ESP_HF_PEER_FEAT_ESCO_S4 (C macro), 479
ESP_HF_PEER_FEAT_EXTERR (C macro), 479
ESP_HF_PEER_FEAT_HF_IND (C macro), 479
ESP_HF_PEER_FEAT_INBAND (C macro), 479
ESP_HF_PEER_FEAT_VREC (C macro), 479
ESP_HF_PEER_FEAT_VREJECT (C macro), 479
ESP_HF_PEER_FEAT_VTAG (C macro), 479
esp_hf_roaming_status_t (C++ enum), 445
esp_hf_roaming_status_t::ESP_HF_ROAMING_STATUS_ACTIVE (C++ enumerator), 445
esp_hf_roaming_status_t::ESP_HF_ROAMING_STATUS_INACTIVE (C++ enumerator), 445
esp_hf_service_type_t (C++ enum), 445
esp_hf_service_type_t::ESP_HF_SERVICE_TYPE_HOME (C++ enumerator), 445
esp_hf_service_type_t::ESP_HF_SERVICE_TYPE_ROAMING (C++ enumerator), 445
esp_hf_subscriber_service_type_t (C++ enum), 447
esp_hf_subscriber_service_type_t::ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX (C++ enumerator), 447
esp_hf_subscriber_service_type_t::ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN (C++ enumerator), 447
esp_hf_subscriber_service_type_t::ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE (C++ enumerator), 447
esp_hf_volume_control_target_t (C++ enum), 443
esp_hf_volume_control_target_t::ESP_HF_VOLUME_CONTROL_TARGET_HF (C++ enumerator), 443
esp_hf_volume_control_target_t::ESP_HF_VOLUME_CONTROL_TARGET_UNKNOWN (C++ enumerator), 443
esp_hf_volume_type_t (C++ enum), 444
esp_hf_volume_type_t::ESP_HF_VOLUME_TYPE_MIC (C++ enumerator), 444
esp_hf_volume_type_t::ESP_HF_VOLUME_TYPE_SPK (C++ enumerator), 444
esp_hf_vr_state_t (C++ enum), 443
esp_hf_vr_state_t::ESP_HF_VR_STATE_DISABLED (C++ enumerator), 443
esp_hf_vr_state_t::ESP_HF_VR_STATE_ENABLED (C++ enumerator), 443
esp_hf_wbs_config_t (C++ enum), 448
esp_hf_wbs_config_t::ESP_HF_WBS_NO (C++ enumerator), 448
esp_hf_wbs_config_t::ESP_HF_WBS_NONE (C++ enumerator), 448
esp_hf_wbs_config_t::ESP_HF_WBS_YES (C++ enumerator), 448
esp_hh_cb_t (C++ type), 504
ESP_HID_CLASS_CDR (C macro), 490
ESP_HID_CLASS_COM (C macro), 490
ESP_HID_CLASS_DGT (C macro), 490
ESP_HID_CLASS_GPD (C macro), 490
ESP_HID_CLASS_JOS (C macro), 490
ESP_HID_CLASS_KBD (C macro), 490

ESP_HID_CLASS_MIC (*C macro*), 490
ESP_HID_CLASS_RMC (*C macro*), 490
ESP_HID_CLASS_SED (*C macro*), 490
ESP_HID_CLASS_UNKNOWN (*C macro*), 490
esp_hidd_app_param_t (*C++ struct*), 489
esp_hidd_app_param_t::desc_list (*C++ member*), 489
esp_hidd_app_param_t::desc_list_len (*C++ member*), 489
esp_hidd_app_param_t::description (*C++ member*), 489
esp_hidd_app_param_t::name (*C++ member*), 489
esp_hidd_app_param_t::provider (*C++ member*), 489
esp_hidd_app_param_t::subclass (*C++ member*), 489
esp_hidd_boot_report_id_t (*C++ enum*), 492
esp_hidd_boot_report_id_t::ESP_HIDD_BOOT_REPORT_ID_KEYBOARD (*C++ enumerator*), 492
esp_hidd_boot_report_id_t::ESP_HIDD_BOOT_REPORT_ID_MOUSE (*C++ enumerator*), 492
esp_hidd_cb_event_t (*C++ enum*), 492
esp_hidd_cb_event_t::ESP_HIDD_API_ERR_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_CLOSE_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_DEINIT_EVT (*C++ enumerator*), 492
esp_hidd_cb_event_t::ESP_HIDD_GET_REPORT_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_INIT_EVT (*C++ enumerator*), 492
esp_hidd_cb_event_t::ESP_HIDD_INTR_DATA_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_OPEN_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_REGISTER_APP_EVT (*C++ enumerator*), 492
esp_hidd_cb_event_t::ESP_HIDD_REPORT_ERR_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_SEND_REPORT_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_SET_PROTOCOL_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_SET_REPORT_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_UNREGISTER_APP_EVT (*C++ enumerator*), 493
esp_hidd_cb_event_t::ESP_HIDD_VC_UNPLUG_EVT (*C++ enumerator*), 493
esp_hidd_cb_param_t (*C++ union*), 484
esp_hidd_cb_param_t::close (*C++ member*), 485
esp_hidd_cb_param_t::deinit (*C++ member*), 484
esp_hidd_cb_param_t::get_report (*C++ member*), 485
esp_hidd_cb_param_t::hidd_close_evt_param (*C++ struct*), 485
esp_hidd_cb_param_t::hidd_close_evt_param::conn_s (*C++ member*), 485
esp_hidd_cb_param_t::hidd_close_evt_param::status (*C++ member*), 485
esp_hidd_cb_param_t::hidd_deinit_evt_param (*C++ struct*), 485
esp_hidd_cb_param_t::hidd_deinit_evt_param::statu (*C++ member*), 485
esp_hidd_cb_param_t::hidd_get_report_evt_param (*C++ struct*), 486
esp_hidd_cb_param_t::hidd_get_report_evt_param::b (*C++ member*), 486
esp_hidd_cb_param_t::hidd_get_report_evt_param::r (*C++ member*), 486
esp_hidd_cb_param_t::hidd_get_report_evt_param::r (*C++ member*), 486
esp_hidd_cb_param_t::hidd_init_evt_param (*C++ struct*), 486
esp_hidd_cb_param_t::hidd_init_evt_param::status (*C++ member*), 486
esp_hidd_cb_param_t::hidd_intr_data_evt_param (*C++ struct*), 486
esp_hidd_cb_param_t::hidd_intr_data_evt_param::da (*C++ member*), 486
esp_hidd_cb_param_t::hidd_intr_data_evt_param::le (*C++ member*), 486
esp_hidd_cb_param_t::hidd_intr_data_evt_param::re (*C++ member*), 486
esp_hidd_cb_param_t::hidd_open_evt_param (*C++ struct*), 486
esp_hidd_cb_param_t::hidd_open_evt_param::bd_addr (*C++ member*), 487
esp_hidd_cb_param_t::hidd_open_evt_param::conn_st (*C++ member*), 486
esp_hidd_cb_param_t::hidd_open_evt_param::status (*C++ member*), 486
esp_hidd_cb_param_t::hidd_register_app_evt_param (*C++ struct*), 487
esp_hidd_cb_param_t::hidd_register_app_evt_param: (*C++ member*), 487
esp_hidd_cb_param_t::hidd_register_app_evt_param: (*C++ member*), 487
esp_hidd_cb_param_t::hidd_register_app_evt_param: (*C++ member*), 487
esp_hidd_cb_param_t::hidd_report_err_evt_param (*C++ struct*), 487
esp_hidd_cb_param_t::hidd_report_err_evt_param::r (*C++ member*), 487
esp_hidd_cb_param_t::hidd_report_err_evt_param::s (*C++ member*), 487
esp_hidd_cb_param_t::hidd_send_report_evt_param (*C++ struct*), 487
esp_hidd_cb_param_t::hidd_send_report_evt_param:: (*C++ member*), 487
esp_hidd_cb_param_t::hidd_send_report_evt_param:: (*C++ member*), 487

esp_hidd_cb_param_t::hidd_send_report_evt_param::conn_type (C++ member), 487
esp_hidd_cb_param_t::hidd_send_report_evt_param::data (C++ member), 487
esp_hidd_cb_param_t::hidd_set_protocol_evt_param (C++ struct), 488
esp_hidd_cb_param_t::hidd_set_protocol_evt_param::protocol_mode (C++ member), 488
esp_hidd_cb_param_t::hidd_set_report_evt_param (C++ struct), 488
esp_hidd_cb_param_t::hidd_set_report_evt_param::data (C++ member), 488
esp_hidd_cb_param_t::hidd_set_report_evt_param::len (C++ member), 488
esp_hidd_cb_param_t::hidd_set_report_evt_param::report_type (C++ member), 488
esp_hidd_cb_param_t::hidd_set_report_evt_param::reserved (C++ member), 488
esp_hidd_cb_param_t::hidd_unregister_app_evt_param (C++ struct), 488
esp_hidd_cb_param_t::hidd_unregister_app_evt_param::status (C++ member), 488
esp_hidd_cb_param_t::hidd_vc_unplug_param (C++ struct), 488
esp_hidd_cb_param_t::hidd_vc_unplug_param::conn_status (C++ member), 488
esp_hidd_cb_param_t::hidd_vc_unplug_param::reserved (C++ member), 488
esp_hidd_cb_param_t::init (C++ member), 484
esp_hidd_cb_param_t::intr_data (C++ member), 485
esp_hidd_cb_param_t::open (C++ member), 485
esp_hidd_cb_param_t::register_app (C++ member), 484
esp_hidd_cb_param_t::report_err (C++ member), 485
esp_hidd_cb_param_t::send_report (C++ member), 485
esp_hidd_cb_param_t::set_protocol (C++ member), 485
esp_hidd_cb_param_t::set_report (C++ member), 485
esp_hidd_cb_param_t::unregister_app (C++ member), 484
esp_hidd_cb_param_t::vc_unplug (C++ member), 485
esp_hidd_connection_state_t (C++ enum), 491
esp_hidd_connection_state_t::ESP_HIDD_CONN_STATE_CONNECTED (C++ enumerator), 491
esp_hidd_connection_state_t::ESP_HIDD_CONN_STATE_CONNECTING (C++ enumerator), 491
esp_hidd_connection_state_t::ESP_HIDD_CONN_STATE_DISCONNECTED (C++ enumerator), 491
esp_hidd_connection_state_t::ESP_HIDD_CONN_STATE_DISCONNECTING (C++ enumerator), 491
esp_hidd_connection_state_t::ESP_HIDD_CONN_STATE_UNKNOWN (C++ enumerator), 492
esp_hidd_handshake_error_t (C++ enum), 490
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_ERR_FATAL (C++ enumerator), 491
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_ERR_INTERNAL (C++ enumerator), 491
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_ERR_INVALID_PARAM (C++ enumerator), 491
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_ERR_NOT_READY (C++ enumerator), 491
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_ERR_UNKNOWN (C++ enumerator), 490
esp_hidd_handshake_error_t::ESP_HID_PAR_HANDSHAKE_RSP_SUCCESS (C++ enumerator), 490
esp_hidd_protocol_mode_t (C++ enum), 492
esp_hidd_protocol_mode_t::ESP_HIDD_BOOT_MODE (C++ enumerator), 492
esp_hidd_protocol_mode_t::ESP_HIDD_REPORT_MODE (C++ enumerator), 492
esp_hidd_protocol_mode_t::ESP_HIDD_UNSUPPORTED_MODE (C++ enumerator), 492
esp_hidd_qos_param_t (C++ struct), 489
esp_hidd_qos_param_t::access_latency (C++ member), 489
esp_hidd_qos_param_t::delay_variation (C++ member), 489
esp_hidd_qos_param_t::peak_bandwidth (C++ member), 489
esp_hidd_qos_param_t::service_type (C++ member), 489
esp_hidd_qos_param_t::token_bucket_size (C++ member), 489
esp_hidd_qos_param_t::token_rate (C++ member), 489
esp_hidd_report_type_t (C++ enum), 491
esp_hidd_report_type_t::ESP_HIDD_REPORT_TYPE_FEATURE (C++ enumerator), 491
esp_hidd_report_type_t::ESP_HIDD_REPORT_TYPE_INPUT (C++ enumerator), 491
esp_hidd_report_type_t::ESP_HIDD_REPORT_TYPE_INTR (C++ enumerator), 491
esp_hidd_report_type_t::ESP_HIDD_REPORT_TYPE_OTHER (C++ enumerator), 491
esp_hidd_report_type_t::ESP_HIDD_REPORT_TYPE_OUTPUT (C++ enumerator), 491
esp_hidd_status_t (C++ enum), 493
esp_hidd_status_t::ESP_HIDD_BUSY (C++ enumerator), 493
esp_hidd_status_t::ESP_HIDD_ERROR (C++ enumerator), 493
esp_hidd_status_t::ESP_HIDD_NEED_DEINIT (C++ enumerator), 494
esp_hidd_status_t::ESP_HIDD_NEED_DEREG (C++ enumerator), 494

esp_hidd_status_t::ESP_HIDD_NEED_INIT (C++ enumerator), 494
esp_hidd_status_t::ESP_HIDD_NEED_REG (C++ enumerator), 494
esp_hidd_status_t::ESP_HIDD_NO_CONNECTION (C++ enumerator), 494
esp_hidd_status_t::ESP_HIDD_NO_DATA (C++ enumerator), 493
esp_hidd_status_t::ESP_HIDD_NO_RES (C++ enumerator), 493
esp_hidd_status_t::ESP_HIDD_SUCCESS (C++ enumerator), 493
esp_hidh_cb_event_t (C++ enum), 506
esp_hidh_cb_event_t::ESP_HIDH_ADD_DEV_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_CLOSE_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_DATA_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_DATA_IND_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_DEINIT_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_GET_DSCP_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_GET_IDLE_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_GET_PROTO_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_GET_RPT_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_INIT_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_OPEN_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_RMV_DEV_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_SET_IDLE_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_SET_INFO_EVT (C++ enumerator), 508
esp_hidh_cb_event_t::ESP_HIDH_SET_PROTO_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_SET_RPT_EVT (C++ enumerator), 507
esp_hidh_cb_event_t::ESP_HIDH_VC_UNPLUG_EVT (C++ enumerator), 507
esp_hidh_cb_param_t (C++ union), 496
esp_hidh_cb_param_t::add_dev (C++ member), 497
esp_hidh_cb_param_t::close (C++ member), 497
esp_hidh_cb_param_t::data_ind (C++ member), 497
esp_hidh_cb_param_t::deinit (C++ member), 497
esp_hidh_cb_param_t::dscp (C++ member), 497
esp_hidh_cb_param_t::get_idle (C++ member), 497
esp_hidh_cb_param_t::get_proto (C++ member), 497
esp_hidh_cb_param_t::get_rpt (C++ member), 497
esp_hidh_cb_param_t::hidh_add_dev_evt_param (C++ struct), 498
esp_hidh_cb_param_t::hidh_add_dev_evt_param::bd_a (C++ member), 498
esp_hidh_cb_param_t::hidh_add_dev_evt_param::hand (C++ member), 498
esp_hidh_cb_param_t::hidh_add_dev_evt_param::stat (C++ member), 498
esp_hidh_cb_param_t::hidh_close_evt_param (C++ struct), 498
esp_hidh_cb_param_t::hidh_close_evt_param::conn_s (C++ member), 498
esp_hidh_cb_param_t::hidh_close_evt_param::handle (C++ member), 498
esp_hidh_cb_param_t::hidh_close_evt_param::reason (C++ member), 498
esp_hidh_cb_param_t::hidh_close_evt_param::status (C++ member), 498
esp_hidh_cb_param_t::hidh_data_ind_evt_param (C++ struct), 498
esp_hidh_cb_param_t::hidh_data_ind_evt_param::dat (C++ member), 499
esp_hidh_cb_param_t::hidh_data_ind_evt_param::han (C++ member), 498
esp_hidh_cb_param_t::hidh_data_ind_evt_param::len (C++ member), 499
esp_hidh_cb_param_t::hidh_data_ind_evt_param::pro (C++ member), 498
esp_hidh_cb_param_t::hidh_data_ind_evt_param::sta (C++ member), 498
esp_hidh_cb_param_t::hidh_get_dscp_evt_param (C++ struct), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::add (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::ctr (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::dl_ (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::dsc (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::han (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::pro (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::ssr (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::ssr (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::sta (C++ member), 499
esp_hidh_cb_param_t::hidh_get_dscp_evt_param::ven (C++ member), 499

esp_hidh_cb_param_t::hidh_get_dscp_evt_param::dscp (C++ member), 499
esp_hidh_cb_param_t::hidh_get_idle_evt_param (C++ struct), 499
esp_hidh_cb_param_t::hidh_get_idle_evt_param::handle (C++ member), 500
esp_hidh_cb_param_t::hidh_get_idle_evt_param::idle_rate (C++ member), 500
esp_hidh_cb_param_t::hidh_get_idle_evt_param::status (C++ member), 500
esp_hidh_cb_param_t::hidh_get_proto_evt_param (C++ struct), 500
esp_hidh_cb_param_t::hidh_get_proto_evt_param::handle (C++ member), 500
esp_hidh_cb_param_t::hidh_get_proto_evt_param::proto_mode (C++ member), 500
esp_hidh_cb_param_t::hidh_get_proto_evt_param::status (C++ member), 500
esp_hidh_cb_param_t::hidh_get_rpt_evt_param (C++ struct), 500
esp_hidh_cb_param_t::hidh_get_rpt_evt_param::data (C++ member), 500
esp_hidh_cb_param_t::hidh_get_rpt_evt_param::handle (C++ member), 500
esp_hidh_cb_param_t::hidh_get_rpt_evt_param::len (C++ member), 500
esp_hidh_cb_param_t::hidh_get_rpt_evt_param::status (C++ member), 500
esp_hidh_cb_param_t::hidh_init_evt_param (C++ struct), 500
esp_hidh_cb_param_t::hidh_init_evt_param::status (C++ member), 501
esp_hidh_cb_param_t::hidh_open_evt_param (C++ struct), 501
esp_hidh_cb_param_t::hidh_open_evt_param::bd_addr (C++ member), 501
esp_hidh_cb_param_t::hidh_open_evt_param::conn_status (C++ member), 501
esp_hidh_cb_param_t::hidh_open_evt_param::handle (C++ member), 501
esp_hidh_cb_param_t::hidh_open_evt_param::hid_dscp (C++ member), 501
esp_hidh_cb_param_t::hidh_open_evt_param::status (C++ member), 501
esp_hidh_cb_param_t::hidh_rmv_dev_evt_param (C++ struct), 501
esp_hidh_cb_param_t::hidh_rmv_dev_evt_param::bd_addr (C++ member), 501
esp_hidh_cb_param_t::hidh_rmv_dev_evt_param::handle (C++ member), 501
esp_hidh_cb_param_t::hidh_rmv_dev_evt_param::status (C++ member), 501
esp_hidh_cb_param_t::hidh_send_data_evt_param (C++ struct), 501
esp_hidh_cb_param_t::hidh_send_data_evt_param::handle (C++ member), 501
esp_hidh_cb_param_t::hidh_send_data_evt_param::reason (C++ member), 502
esp_hidh_cb_param_t::hidh_send_data_evt_param::st (C++ member), 501
esp_hidh_cb_param_t::hidh_set_idle_evt_param (C++ struct), 502
esp_hidh_cb_param_t::hidh_set_idle_evt_param::han (C++ member), 502
esp_hidh_cb_param_t::hidh_set_idle_evt_param::sta (C++ member), 502
esp_hidh_cb_param_t::hidh_set_info_evt_param (C++ struct), 502
esp_hidh_cb_param_t::hidh_set_info_evt_param::bd_ (C++ member), 502
esp_hidh_cb_param_t::hidh_set_info_evt_param::han (C++ member), 502
esp_hidh_cb_param_t::hidh_set_info_evt_param::sta (C++ member), 502
esp_hidh_cb_param_t::hidh_set_proto_evt_param (C++ struct), 502
esp_hidh_cb_param_t::hidh_set_proto_evt_param::ha (C++ member), 502
esp_hidh_cb_param_t::hidh_set_proto_evt_param::st (C++ member), 502
esp_hidh_cb_param_t::hidh_set_rpt_evt_param (C++ struct), 502
esp_hidh_cb_param_t::hidh_set_rpt_evt_param::hand (C++ member), 503
esp_hidh_cb_param_t::hidh_set_rpt_evt_param::stat (C++ member), 503
esp_hidh_cb_param_t::hidh_uninit_evt_param (C++ struct), 503
esp_hidh_cb_param_t::hidh_uninit_evt_param::statu (C++ member), 503
esp_hidh_cb_param_t::hidh_unplug_evt_param (C++ struct), 503
esp_hidh_cb_param_t::hidh_unplug_evt_param::conn_ (C++ member), 503
esp_hidh_cb_param_t::hidh_unplug_evt_param::handl (C++ member), 503
esp_hidh_cb_param_t::hidh_unplug_evt_param::statu (C++ member), 503
esp_hidh_cb_param_t::init (C++ member), 497
esp_hidh_cb_param_t::open (C++ member), 497
esp_hidh_cb_param_t::rmv_dev (C++ member), 497
esp_hidh_cb_param_t::send_data (C++ member), 497
esp_hidh_cb_param_t::set_idle (C++ member), 497
esp_hidh_cb_param_t::set_info (C++ member), 497
esp_hidh_cb_param_t::set_proto (C++ member), 497
esp_hidh_cb_param_t::set_rpt (C++ member), 497
esp_hidh_cb_param_t::unplug (C++ member), 497

esp_hidh_connection_state_t (*C++ enum*), 504
esp_hidh_connection_state_t::ESP_HIDH_CONN_STATE_CONNECTED (*C++ enumerator*), 504
esp_hidh_connection_state_t::ESP_HIDH_CONN_STATE_CONNECTING (*C++ enumerator*), 504
esp_hidh_connection_state_t::ESP_HIDH_CONN_STATE_DISCONNECTED (*C++ enumerator*), 504
esp_hidh_connection_state_t::ESP_HIDH_CONN_STATE_DISCONNECTING (*C++ enumerator*), 504
esp_hidh_connection_state_t::ESP_HIDH_CONN_STATE_UNKNOWN (*C++ enumerator*), 504
esp_hidh_dev_app_id_t (*C++ enum*), 508
esp_hidh_dev_app_id_t::ESP_HIDH_APP_ID_GAMEPAD (*C++ enumerator*), 508
esp_hidh_dev_app_id_t::ESP_HIDH_APP_ID_JOYSTICK (*C++ enumerator*), 508
esp_hidh_dev_app_id_t::ESP_HIDH_APP_ID_KEYBOARD (*C++ enumerator*), 508
esp_hidh_dev_app_id_t::ESP_HIDH_APP_ID_MOUSE (*C++ enumerator*), 508
esp_hidh_dev_app_id_t::ESP_HIDH_APP_ID_REMOTE_CONTROL (*C++ enumerator*), 508
esp_hidh_dev_attr_t (*C++ enum*), 508
esp_hidh_dev_attr_t::ESP_HIDH_DEV_ATTR_NORMALLY_CONNECTABLE (*C++ enumerator*), 508
esp_hidh_dev_attr_t::ESP_HIDH_DEV_ATTR_RECONNECT_INITIATE (*C++ enumerator*), 508
esp_hidh_dev_attr_t::ESP_HIDH_DEV_ATTR_VIRTUAL_CABLE (*C++ enumerator*), 508
esp_hidh_hid_info_t (*C++ struct*), 503
esp_hidh_hid_info_t::app_id (*C++ member*), 503
esp_hidh_hid_info_t::attr_mask (*C++ member*), 503
esp_hidh_hid_info_t::ctry_code (*C++ member*), 504
esp_hidh_hid_info_t::dl_len (*C++ member*), 504
esp_hidh_hid_info_t::dsc_list (*C++ member*), 504
esp_hidh_hid_info_t::product_id (*C++ member*), 504
esp_hidh_hid_info_t::sub_class (*C++ member*), 503
esp_hidh_hid_info_t::vendor_id (*C++ member*), 503
esp_hidh_hid_info_t::version (*C++ member*), 504
esp_hidh_protocol_mode_t (*C++ enum*), 506
esp_hidh_protocol_mode_t::ESP_HIDH_BOOT_MODE (*C++ enumerator*), 506
esp_hidh_protocol_mode_t::ESP_HIDH_REPORT_MODE (*C++ enumerator*), 506
esp_hidh_protocol_mode_t::ESP_HIDH_UNSUPPORTED_MODE (*C++ enumerator*), 506
esp_hidh_report_type_t (*C++ enum*), 506
esp_hidh_report_type_t::ESP_HIDH_REPORT_TYPE_FEATURE (*C++ enumerator*), 506
esp_hidh_report_type_t::ESP_HIDH_REPORT_TYPE_INPUT (*C++ enumerator*), 506
esp_hidh_report_type_t::ESP_HIDH_REPORT_TYPE_OTHER (*C++ enumerator*), 506
esp_hidh_report_type_t::ESP_HIDH_REPORT_TYPE_OUTPUT (*C++ enumerator*), 506
esp_hidh_status_t (*C++ enum*), 505
esp_hidh_status_t::ESP_HIDH_BUSY (*C++ enumerator*), 506
esp_hidh_status_t::ESP_HIDH_ERR (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_AUTH_FAILED (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_DB_FULL (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_HDL (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_NO_RES (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_PROTO (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_SDP (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_SEC (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_ERR_TOD_UNSPT (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_HS_ERROR (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_HS_HID_NOT_READY (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_HS_INVALID_PARAM (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_HS_INVALID_RPT_ID (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_HS_TRANS_NOT_SPT (*C++ enumerator*), 505
esp_hidh_status_t::ESP_HIDH_NEED_DEINIT (*C++ enumerator*), 506
esp_hidh_status_t::ESP_HIDH_NEED_INIT (*C++ enumerator*), 506
esp_hidh_status_t::ESP_HIDH_NO_CONNECTION (*C++ enumerator*), 506
esp_hidh_status_t::ESP_HIDH_NO_DATA (*C++ enumerator*), 506
esp_hidh_status_t::ESP_HIDH_OK (*C++ enumerator*), 505
esp_himem_alloc (*C++ function*), 2150
esp_himem_alloc_map_range (*C++ function*), 2150
ESP_HIMEM_BLKSZ (*C macro*), 2151
esp_himem_free (*C++ function*), 2150
esp_himem_free_map_range (*C++ function*), 2151
esp_himem_get_free_size (*C++ function*), 2151
esp_himem_get_phys_size (*C++ function*),

2151
esp_himem_handle_t (C++ type), 2151
esp_himem_map (C++ function), 2150
ESP_HIMEM_MAPFLAG_RO (C macro), 2151
esp_himem_rangehandle_t (C++ type), 2151
esp_himem_reserved_area_size (C++ function), 2151
esp_himem_unmap (C++ function), 2151
esp_http_client_add_auth (C++ function), 162
esp_http_client_auth_type_t (C++ enum), 169
esp_http_client_auth_type_t::HTTP_AUTH_TYPE_BASIC (C++ enumerator), 169
esp_http_client_auth_type_t::HTTP_AUTH_TYPE_DIGEST (C++ enumerator), 169
esp_http_client_auth_type_t::HTTP_AUTH_TYPE_NONE (C++ enumerator), 169
esp_http_client_cleanup (C++ function), 161
esp_http_client_close (C++ function), 161
esp_http_client_config_t (C++ struct), 163
esp_http_client_config_t::auth_type (C++ member), 164
esp_http_client_config_t::buffer_size (C++ member), 165
esp_http_client_config_t::buffer_size_tx (C++ member), 165
esp_http_client_config_t::cert_len (C++ member), 164
esp_http_client_config_t::cert_pem (C++ member), 164
esp_http_client_config_t::client_cert_len (C++ member), 164
esp_http_client_config_t::client_cert_pem (C++ member), 164
esp_http_client_config_t::client_key_len (C++ member), 164
esp_http_client_config_t::client_key_password (C++ member), 164
esp_http_client_config_t::client_key_password_len (C++ member), 165
esp_http_client_config_t::client_key_pem (C++ member), 164
esp_http_client_config_t::crt_bundle_attach (C++ member), 165
esp_http_client_config_t::disable_auto_redirect (C++ member), 165
esp_http_client_config_t::event_handler (C++ member), 165
esp_http_client_config_t::host (C++ member), 164
esp_http_client_config_t::if_name (C++ member), 166
esp_http_client_config_t::is_async (C++ member), 165
esp_http_client_config_t::keep_alive_count (C++ member), 166
esp_http_client_config_t::keep_alive_enable (C++ member), 166
esp_http_client_config_t::keep_alive_idle (C++ member), 166
esp_http_client_config_t::keep_alive_interval (C++ member), 166
esp_http_client_config_t::max_authorization_retri (C++ member), 165
esp_http_client_config_t::max_redirection_count (C++ member), 165
esp_http_client_config_t::method (C++ member), 165
esp_http_client_config_t::password (C++ member), 164
esp_http_client_config_t::path (C++ member), 164
esp_http_client_config_t::port (C++ member), 164
esp_http_client_config_t::query (C++ member), 164
esp_http_client_config_t::skip_cert_common_name_c (C++ member), 165
esp_http_client_config_t::timeout_ms (C++ member), 165
esp_http_client_config_t::transport_type (C++ member), 165
esp_http_client_config_t::url (C++ member), 164
esp_http_client_config_t::use_global_ca_store (C++ member), 165
esp_http_client_config_t::user_agent (C++ member), 165
esp_http_client_config_t::user_data (C++ member), 165
esp_http_client_config_t::username (C++ member), 164
esp_http_client_delete_header (C++ function), 160
esp_http_client_event (C++ struct), 163
esp_http_client_event::client (C++ member), 163
esp_http_client_event::data (C++ member), 163
esp_http_client_event::data_len (C++ member), 163
esp_http_client_event::event_id (C++ member), 163
esp_http_client_event::header_key (C++ member), 163
esp_http_client_event::header_value (C++ member), 163
esp_http_client_event::user_data (C++ member), 163
esp_http_client_event_handle_t (C++ type), 167
esp_http_client_event_id_t (C++ enum), 167
esp_http_client_event_id_t::HTTP_EVENT_DISCONNECT (C++ enumerator), 167

esp_http_client_event_id_t::HTTP_EVENT_ERROR (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_HEADER_SENT (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_HEADERS_SENT (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_ON_CONNECTED (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_ON_DATA (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_ON_FINISH (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_ON_HEADER (*C++ enumerator*), 167
esp_http_client_event_id_t::HTTP_EVENT_REDIRECT (*C++ enumerator*), 167
esp_http_client_event_t (*C++ type*), 167
esp_http_client_fetch_headers (*C++ function*), 160
esp_http_client_flush_response (*C++ function*), 162
esp_http_client_get_chunk_length (*C++ function*), 163
esp_http_client_get_content_length (*C++ function*), 161
esp_http_client_get_errno (*C++ function*), 159
esp_http_client_get_header (*C++ function*), 158
esp_http_client_get_password (*C++ function*), 159
esp_http_client_get_post_field (*C++ function*), 158
esp_http_client_get_status_code (*C++ function*), 161
esp_http_client_get_transport_type (*C++ function*), 162
esp_http_client_get_url (*C++ function*), 163
esp_http_client_get_username (*C++ function*), 159
esp_http_client_handle_t (*C++ type*), 167
esp_http_client_init (*C++ function*), 157
esp_http_client_is_chunked_response (*C++ function*), 161
esp_http_client_is_complete_data_received (*C++ function*), 162
esp_http_client_method_t (*C++ enum*), 168
esp_http_client_method_t::HTTP_METHOD_COPY (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_DELETE (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_GET (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_HEAD (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_LOCK (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_MAX (*C++ enumerator*), 169
esp_http_client_method_t::HTTP_METHOD_MKCOL (*C++ enumerator*), 169
esp_http_client_method_t::HTTP_METHOD_MOVE (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_NOTIFY (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_OPTIONS (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_PATCH (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_POST (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_PROPFIND (*C++ enumerator*), 169
esp_http_client_method_t::HTTP_METHOD_PROPPATCH (*C++ enumerator*), 169
esp_http_client_method_t::HTTP_METHOD_PUT (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_SUBSCRIBE (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_UNLOCK (*C++ enumerator*), 168
esp_http_client_method_t::HTTP_METHOD_UNSUBSCRIBE (*C++ enumerator*), 168
esp_http_client_open (*C++ function*), 160
esp_http_client_perform (*C++ function*), 157
esp_http_client_read (*C++ function*), 161
esp_http_client_read_response (*C++ function*), 162
esp_http_client_set_authtype (*C++ function*), 159
esp_http_client_set_header (*C++ function*), 158
esp_http_client_set_method (*C++ function*), 160
esp_http_client_set_password (*C++ function*), 159
esp_http_client_set_post_field (*C++ function*), 158
esp_http_client_set_redirection (*C++ function*), 162
esp_http_client_set_timeout_ms (*C++ function*), 160
esp_http_client_set_url (*C++ function*), 158
esp_http_client_set_username (*C++ function*), 159
esp_http_client_transport_t (*C++ enum*), 167
esp_http_client_transport_t::HTTP_TRANSPORT_OVER_SSL (*C++ enumerator*), 168
esp_http_client_transport_t::HTTP_TRANSPORT_OVER_TCP (*C++ enumerator*), 167
esp_http_client_transport_t::HTTP_TRANSPORT_UNKNOWN (*C++ enumerator*), 167
esp_http_client_write (*C++ function*), 160
esp_http_server_event_data (*C++ struct*), 216

esp_http_server_event_data::data_len (*C++ member*), 216
esp_http_server_event_data::fd (*C++ member*), 216
esp_http_server_event_id_t (*C++ enum*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_DISCONNECTED (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_ERROR (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_HEADERS_SENT (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_ON_CONNECTED (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_ON_DATA (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_ON_HEADER (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_SENT_DATA (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_START (*C++ enumerator*), 225
esp_http_server_event_id_t::HTTP_SERVER_EVENT_STOP (*C++ enumerator*), 225
ESP_HTTPD_DEF_CTRL_PORT (*C macro*), 221
esp_https_ota (*C++ function*), 1875
esp_https_ota_abort (*C++ function*), 1876
esp_https_ota_begin (*C++ function*), 1875
esp_https_ota_config_t (*C++ struct*), 1877
esp_https_ota_config_t::bulk_flash_erase (*C++ member*), 1878
esp_https_ota_config_t::http_client_init_cb (*C++ member*), 1878
esp_https_ota_config_t::http_config (*C++ member*), 1878
esp_https_ota_config_t::max_http_request_size (*C++ member*), 1878
esp_https_ota_config_t::partial_http_download (*C++ member*), 1878
esp_https_ota_event_t (*C++ enum*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_ABORT (*C++ enumerator*), 1879
esp_https_ota_event_t::ESP_HTTPS_OTA_CONNECTED (*C++ enumerator*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_DECRYPT_CB (*C++ enumerator*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_FINISH (*C++ enumerator*), 1879
esp_https_ota_event_t::ESP_HTTPS_OTA_GET_IMG_DESC (*C++ enumerator*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_START (*C++ enumerator*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION (*C++ enumerator*), 1879
esp_https_ota_event_t::ESP_HTTPS_OTA_VERIFY_CHIP_ID (*C++ enumerator*), 1878
esp_https_ota_event_t::ESP_HTTPS_OTA_WRITE_FLASH (*C++ enumerator*), 1879
esp_https_ota_finish (*C++ function*), 1876
esp_https_ota_get_image_len_read (*C++ function*), 1877
esp_https_ota_get_image_size (*C++ function*), 1877
esp_https_ota_get_img_desc (*C++ function*), 1877
esp_https_ota_handle_t (*C++ type*), 1878
esp_https_ota_is_complete_data_received (*C++ function*), 1876
esp_https_ota_perform (*C++ function*), 1875
esp_https_server_user_cb (*C++ type*), 228
esp_https_server_user_cb_arg (*C++ struct*), 227
esp_https_server_user_cb_arg::tls (*C++ member*), 227
esp_https_server_user_cb_arg::user_cb_state (*C++ member*), 227
esp_https_server_user_cb_arg_t (*C++ type*), 228
ESP_IDF_VERSION (*C macro*), 2086
ESP_IDF_VERSION_MAJOR (*C macro*), 2086
ESP_IDF_VERSION_MINOR (*C macro*), 2086
ESP_IDF_VERSION_PATCH (*C macro*), 2086
ESP_IDF_VERSION_VAL (*C macro*), 2086
esp_image_flash_size_t (*C++ enum*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_128MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_16MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_1MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_2MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_32MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_4MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_64MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_8MB (*C++ enumerator*), 1831
esp_image_flash_size_t::ESP_IMAGE_FLASH_SIZE_MAX (*C++ enumerator*), 1831
ESP_IMAGE_HEADER_MAGIC (*C macro*), 1829
esp_image_header_t (*C++ struct*), 1828
esp_image_header_t::chip_id (*C++ member*), 1828
esp_image_header_t::entry_addr (*C++ member*), 1828
esp_image_header_t::hash_appended (*C++ member*), 1829
esp_image_header_t::magic (*C++ member*), 1828
esp_image_header_t::max_chip_rev_full (*C++ member*), 1829
esp_image_header_t::min_chip_rev (*C++ member*), 1828

esp_image_header_t::min_chip_rev_full (C++ member), 1829
esp_image_header_t::reserved (C++ member), 1829
esp_image_header_t::segment_count (C++ member), 1828
esp_image_header_t::spi_mode (C++ member), 1828
esp_image_header_t::spi_pin_drv (C++ member), 1828
esp_image_header_t::spi_size (C++ member), 1828
esp_image_header_t::spi_speed (C++ member), 1828
esp_image_header_t::wp_pin (C++ member), 1828
ESP_IMAGE_MAX_SEGMENTS (C macro), 1829
esp_image_segment_header_t (C++ struct), 1829
esp_image_segment_header_t::data_len (C++ member), 1829
esp_image_segment_header_t::load_addr (C++ member), 1829
esp_image_spi_freq_t (C++ enum), 1830
esp_image_spi_freq_t::ESP_IMAGE_SPI_SPEED_DIV_1 (C++ enumerator), 1830
esp_image_spi_freq_t::ESP_IMAGE_SPI_SPEED_DIV_2 (C++ enumerator), 1830
esp_image_spi_freq_t::ESP_IMAGE_SPI_SPEED_DIV_3 (C++ enumerator), 1830
esp_image_spi_freq_t::ESP_IMAGE_SPI_SPEED_DIV_4 (C++ enumerator), 1830
esp_image_spi_mode_t (C++ enum), 1830
esp_image_spi_mode_t::ESP_IMAGE_SPI_MODE_DIO (C++ enumerator), 1830
esp_image_spi_mode_t::ESP_IMAGE_SPI_MODE_DOUT (C++ enumerator), 1830
esp_image_spi_mode_t::ESP_IMAGE_SPI_MODE_FAST_READ (C++ enumerator), 1830
esp_image_spi_mode_t::ESP_IMAGE_SPI_MODE_QIO (C++ enumerator), 1830
esp_image_spi_mode_t::ESP_IMAGE_SPI_MODE_SLOW_READ (C++ enumerator), 1830
esp_intr_alloc (C++ function), 2069
esp_intr_alloc_intrstatus (C++ function), 2070
ESP_INTR_DISABLE (C macro), 2073
esp_intr_disable (C++ function), 2071
esp_intr_disable_source (C++ function), 2072
ESP_INTR_ENABLE (C macro), 2073
esp_intr_enable (C++ function), 2071
esp_intr_enable_source (C++ function), 2072
ESP_INTR_FLAG_EDGE (C macro), 2072
ESP_INTR_FLAG_HIGH (C macro), 2073
ESP_INTR_FLAG_INTRDISABLED (C macro), 2072
ESP_INTR_FLAG_IRAM (C macro), 2072
ESP_INTR_FLAG_LEVEL1 (C macro), 2072
ESP_INTR_FLAG_LEVEL2 (C macro), 2072
ESP_INTR_FLAG_LEVEL3 (C macro), 2072
ESP_INTR_FLAG_LEVEL4 (C macro), 2072
ESP_INTR_FLAG_LEVEL5 (C macro), 2072
ESP_INTR_FLAG_LEVEL6 (C macro), 2072
ESP_INTR_FLAG_LEVELMASK (C macro), 2073
ESP_INTR_FLAG_LOWMED (C macro), 2073
ESP_INTR_FLAG_NMI (C macro), 2072
ESP_INTR_FLAG_SHARED (C macro), 2072
esp_intr_flags_to_level (C++ function), 2072
esp_intr_free (C++ function), 2070
esp_intr_get_cpu (C++ function), 2071
esp_intr_get_intno (C++ function), 2071
esp_intr_mark_shared (C++ function), 2069
esp_intr_noniram_disable (C++ function), 2071
esp_intr_noniram_enable (C++ function), 2072
esp_intr_reserve (C++ function), 2069
esp_intr_set_in_iram (C++ function), 2071
ESP_IO_CAP_IN (C macro), 295
ESP_IO_CAP_IO (C macro), 295
ESP_IO_CAP_KBDISP (C macro), 295
ESP_IO_CAP_NONE (C macro), 295
ESP_IO_CAP_OUT (C macro), 295
esp_ip4_addr (C++ struct), 1014
esp_ip4_addr1 (C macro), 1014
esp_ip4_addr1_16 (C macro), 1015
esp_ip4_addr2 (C macro), 1015
esp_ip4_addr2_16 (C macro), 1015
esp_ip4_addr3 (C macro), 1015
esp_ip4_addr3_16 (C macro), 1015
esp_ip4_addr4 (C macro), 1015
esp_ip4_addr4_16 (C macro), 1015
esp_ip4_addr::addr (C++ member), 1014
esp_ip4_addr_get_byte (C macro), 1014
esp_ip4_addr_t (C++ type), 1015
esp_ip4addr_aton (C++ function), 1003
ESP_IP4ADDR_INIT (C macro), 1015
esp_ip4addr_ntoa (C++ function), 1002
ESP_IP4TOADDR (C macro), 1015
ESP_IP4TOUINT32 (C macro), 1015
esp_ip6_addr (C++ struct), 1013
esp_ip6_addr::addr (C++ member), 1013
esp_ip6_addr::zone (C++ member), 1013
ESP_IP6_ADDR_BLOCK1 (C macro), 1014
ESP_IP6_ADDR_BLOCK2 (C macro), 1014
ESP_IP6_ADDR_BLOCK3 (C macro), 1014
ESP_IP6_ADDR_BLOCK4 (C macro), 1014
ESP_IP6_ADDR_BLOCK5 (C macro), 1014
ESP_IP6_ADDR_BLOCK6 (C macro), 1014
ESP_IP6_ADDR_BLOCK7 (C macro), 1014
ESP_IP6_ADDR_BLOCK8 (C macro), 1014
esp_ip6_addr_t (C++ type), 1015
esp_ip6_addr_type_t (C++ enum), 1015

esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_GLOBAL (C++ enumerator), 1015
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_IPV4_MAPPED_IPV6 (C++ enumerator), 1016
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_LINK_LOCAL (C++ enumerator), 1015
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_SITE_LOCAL (C++ enumerator), 1015
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_UNIQUE_LOCAL (C++ enumerator), 1016
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_UNKNOWN (C++ enumerator), 1015
ESP_IP6ADDR_INIT (C macro), 1015
esp_ip_addr_t (C++ type), 1015
ESP_IPADDR_TYPE_ANY (C macro), 1015
ESP_IPADDR_TYPE_V4 (C macro), 1015
ESP_IPADDR_TYPE_V6 (C macro), 1015
esp_ipc_call (C++ function), 2065
esp_ipc_call_blocking (C++ function), 2065
esp_ipc_func_t (C++ type), 2065
esp_ipc_isr_asm_call (C++ function), 2066
esp_ipc_isr_asm_call_blocking (C++ function), 2066
esp_ipc_isr_func_t (C++ type), 2067
esp_ipc_isr_release_other_cpu (C++ function), 2066
esp_ipc_isr_stall_abort (C++ function), 2067
esp_ipc_isr_stall_other_cpu (C++ function), 2066
esp_ipc_isr_stall_pause (C++ function), 2067
esp_ipc_isr_stall_resume (C++ function), 2067
esp_lcd_del_i80_bus (C++ function), 1143
esp_lcd_i2c_bus_handle_t (C++ type), 1148
esp_lcd_i80_bus_config_t (C++ struct), 1146
esp_lcd_i80_bus_config_t::bus_width (C++ member), 1146
esp_lcd_i80_bus_config_t::clk_src (C++ member), 1146
esp_lcd_i80_bus_config_t::data_gpio_nums (C++ member), 1146
esp_lcd_i80_bus_config_t::dc_gpio_num (C++ member), 1146
esp_lcd_i80_bus_config_t::max_transfer_bytes (C++ member), 1146
esp_lcd_i80_bus_config_t::psram_trans_align (C++ member), 1146
esp_lcd_i80_bus_config_t::sram_trans_align (C++ member), 1146
esp_lcd_i80_bus_config_t::wr_gpio_num (C++ member), 1146
esp_lcd_i80_bus_handle_t (C++ type), 1148
esp_lcd_new_i80_bus (C++ function), 1143
esp_lcd_new_panel_io_i2c (C++ function), 1143
esp_lcd_new_panel_io_i80 (C++ function), 1143
esp_lcd_new_panel_io_spi (C++ function), 1143
esp_lcd_new_panel_nt35510 (C++ function), 1151
esp_lcd_new_panel_ssd1306 (C++ function), 1151
esp_lcd_new_panel_st7789 (C++ function), 1151
esp_lcd_panel_del (C++ function), 1149
esp_lcd_panel_dev_config_t (C++ struct), 1151
esp_lcd_panel_dev_config_t::bits_per_pixel (C++ member), 1152
esp_lcd_panel_dev_config_t::color_space (C++ member), 1151
esp_lcd_panel_dev_config_t::data_endian (C++ member), 1152
esp_lcd_panel_dev_config_t::flags (C++ member), 1152
esp_lcd_panel_dev_config_t::reset_active_high (C++ member), 1152
esp_lcd_panel_dev_config_t::reset_gpio_num (C++ member), 1151
esp_lcd_panel_dev_config_t::rgb_ele_order (C++ member), 1152
esp_lcd_panel_dev_config_t::rgb_endian (C++ member), 1152
esp_lcd_panel_dev_config_t::vendor_config (C++ member), 1152
esp_lcd_panel_disp_off (C++ function), 1150
esp_lcd_panel_disp_on_off (C++ function), 1150
esp_lcd_panel_draw_bitmap (C++ function), 1149
esp_lcd_panel_handle_t (C++ type), 1141
esp_lcd_panel_init (C++ function), 1148
esp_lcd_panel_invert_color (C++ function), 1150
esp_lcd_panel_io_callbacks_t (C++ struct), 1144
esp_lcd_panel_io_callbacks_t::on_color_trans_done (C++ member), 1144
esp_lcd_panel_io_color_trans_done_cb_t (C++ type), 1148
esp_lcd_panel_io_del (C++ function), 1142
esp_lcd_panel_io_event_data_t (C++ struct), 1144
esp_lcd_panel_io_handle_t (C++ type), 1141
esp_lcd_panel_io_i2c_config_t (C++ struct), 1145
esp_lcd_panel_io_i2c_config_t::control_phase_bytes (C++ member), 1145
esp_lcd_panel_io_i2c_config_t::dc_bit_offset (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::dc_low_on_data (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::dev_addr

(C++ member), 1145
esp_lcd_panel_io_i2c_config_t::disable_control_phase (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::flags (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::lcd_cmd_bits (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::lcd_param_bits (C++ member), 1146
esp_lcd_panel_io_i2c_config_t::on_color_trans_done (C++ member), 1145
esp_lcd_panel_io_i2c_config_t::user_ctx (C++ member), 1145
esp_lcd_panel_io_i80_config_t (C++ struct), 1147
esp_lcd_panel_io_i80_config_t::cs_active_high (C++ member), 1147
esp_lcd_panel_io_i80_config_t::cs_gpio_num (C++ member), 1147
esp_lcd_panel_io_i80_config_t::dc_cmd_level (C++ member), 1147
esp_lcd_panel_io_i80_config_t::dc_data_level (C++ member), 1147
esp_lcd_panel_io_i80_config_t::dc_dummy_level (C++ member), 1147
esp_lcd_panel_io_i80_config_t::dc_idle_level (C++ member), 1147
esp_lcd_panel_io_i80_config_t::dc_levels (C++ member), 1147
esp_lcd_panel_io_i80_config_t::flags (C++ member), 1148
esp_lcd_panel_io_i80_config_t::lcd_cmd_bits (C++ member), 1147
esp_lcd_panel_io_i80_config_t::lcd_param_bits (C++ member), 1147
esp_lcd_panel_io_i80_config_t::on_color_trans_done (C++ member), 1147
esp_lcd_panel_io_i80_config_t::pclk_active_high (C++ member), 1148
esp_lcd_panel_io_i80_config_t::pclk_hz (C++ member), 1147
esp_lcd_panel_io_i80_config_t::pclk_idle_low (C++ member), 1148
esp_lcd_panel_io_i80_config_t::reverse_color_bits (C++ member), 1147
esp_lcd_panel_io_i80_config_t::swap_color_bytes (C++ member), 1148
esp_lcd_panel_io_i80_config_t::trans_queue_depth (C++ member), 1147
esp_lcd_panel_io_i80_config_t::user_ctx (C++ member), 1147
esp_lcd_panel_io_register_event_callbacks (C++ function), 1142
esp_lcd_panel_io_rx_param (C++ function), 1141
esp_lcd_panel_io_spi_config_t (C++ struct), 1144
esp_lcd_panel_io_spi_config_t::cs_gpio_num (C++ member), 1144
esp_lcd_panel_io_spi_config_t::cs_high_active (C++ member), 1145
esp_lcd_panel_io_spi_config_t::dc_gpio_num (C++ member), 1144
esp_lcd_panel_io_spi_config_t::dc_high_on_cmd (C++ member), 1145
esp_lcd_panel_io_spi_config_t::dc_low_on_data (C++ member), 1145
esp_lcd_panel_io_spi_config_t::dc_low_on_param (C++ member), 1145
esp_lcd_panel_io_spi_config_t::flags (C++ member), 1145
esp_lcd_panel_io_spi_config_t::lcd_cmd_bits (C++ member), 1144
esp_lcd_panel_io_spi_config_t::lcd_param_bits (C++ member), 1145
esp_lcd_panel_io_spi_config_t::lsb_first (C++ member), 1145
esp_lcd_panel_io_spi_config_t::octal_mode (C++ member), 1145
esp_lcd_panel_io_spi_config_t::on_color_trans_done (C++ member), 1144
esp_lcd_panel_io_spi_config_t::pclk_hz (C++ member), 1144
esp_lcd_panel_io_spi_config_t::quad_mode (C++ member), 1145
esp_lcd_panel_io_spi_config_t::sio_mode (C++ member), 1145
esp_lcd_panel_io_spi_config_t::spi_mode (C++ member), 1144
esp_lcd_panel_io_spi_config_t::trans_queue_depth (C++ member), 1144
esp_lcd_panel_io_spi_config_t::user_ctx (C++ member), 1144
esp_lcd_panel_io_tx_color (C++ function), 1142
esp_lcd_panel_io_tx_param (C++ function), 1142
esp_lcd_panel_mirror (C++ function), 1149
esp_lcd_panel_reset (C++ function), 1148
esp_lcd_panel_set_gap (C++ function), 1150
esp_lcd_panel_swap_xy (C++ function), 1149
esp_lcd_spi_bus_handle_t (C++ type), 1148
ESP_LE_AUTH_BOND (C macro), 294
ESP_LE_AUTH_NO_BOND (C macro), 294
ESP_LE_AUTH_REQ_BOND_MITM (C macro), 294
ESP_LE_AUTH_REQ_MITM (C macro), 294
ESP_LE_AUTH_REQ_SC_BOND (C macro), 294
ESP_LE_AUTH_REQ_SC_MITM (C macro), 294
ESP_LE_AUTH_REQ_SC_MITM_BOND (C macro), 294
ESP_LE_AUTH_REQ_SC_ONLY (C macro), 294
ESP_LE_KEY_LCSRK (C macro), 294
ESP_LE_KEY_LENC (C macro), 294
ESP_LE_KEY_LID (C macro), 294
ESP_LE_KEY_LLK (C macro), 294
ESP_LE_KEY_NONE (C macro), 293

ESP_LE_KEY_PCSRK (*C macro*), 294
ESP_LE_KEY_PENC (*C macro*), 293
ESP_LE_KEY_PID (*C macro*), 293
ESP_LE_KEY_PLK (*C macro*), 294
esp_light_sleep_start (*C++ function*), 2130
esp_link_key (*C++ type*), 238
esp_local_ctrl_add_property (*C++ function*), 174
esp_local_ctrl_config (*C++ struct*), 177
esp_local_ctrl_config::handlers (*C++ member*), 178
esp_local_ctrl_config::max_properties (*C++ member*), 178
esp_local_ctrl_config::proto_sec (*C++ member*), 178
esp_local_ctrl_config::transport (*C++ member*), 178
esp_local_ctrl_config::transport_config (*C++ member*), 178
esp_local_ctrl_config_t (*C++ type*), 179
esp_local_ctrl_get_property (*C++ function*), 174
esp_local_ctrl_get_transport_ble (*C++ function*), 173
esp_local_ctrl_get_transport_httpd (*C++ function*), 173
esp_local_ctrl_handlers (*C++ struct*), 176
esp_local_ctrl_handlers::get_prop_values (*C++ member*), 176
esp_local_ctrl_handlers::set_prop_values (*C++ member*), 176
esp_local_ctrl_handlers::usr_ctx (*C++ member*), 177
esp_local_ctrl_handlers::usr_ctx_free_fn (*C++ member*), 177
esp_local_ctrl_handlers_t (*C++ type*), 178
esp_local_ctrl_prop (*C++ struct*), 175
esp_local_ctrl_prop::ctx (*C++ member*), 175
esp_local_ctrl_prop::ctx_free_fn (*C++ member*), 175
esp_local_ctrl_prop::flags (*C++ member*), 175
esp_local_ctrl_prop::name (*C++ member*), 175
esp_local_ctrl_prop::size (*C++ member*), 175
esp_local_ctrl_prop::type (*C++ member*), 175
esp_local_ctrl_prop_t (*C++ type*), 178
esp_local_ctrl_prop_val (*C++ struct*), 176
esp_local_ctrl_prop_val::data (*C++ member*), 176
esp_local_ctrl_prop_val::free_fn (*C++ member*), 176
esp_local_ctrl_prop_val::size (*C++ member*), 176
esp_local_ctrl_prop_val_t (*C++ type*), 178
esp_local_ctrl_proto_sec (*C++ enum*), 179
esp_local_ctrl_proto_sec::PROTOCOM_SEC0 (*C++ enumerator*), 179
esp_local_ctrl_proto_sec::PROTOCOM_SEC1 (*C++ enumerator*), 179
esp_local_ctrl_proto_sec::PROTOCOM_SEC2 (*C++ enumerator*), 179
esp_local_ctrl_proto_sec::PROTOCOM_SEC_CUSTOM (*C++ enumerator*), 179
esp_local_ctrl_proto_sec_cfg (*C++ struct*), 177
esp_local_ctrl_proto_sec_cfg::custom_handle (*C++ member*), 177
esp_local_ctrl_proto_sec_cfg::pop (*C++ member*), 177
esp_local_ctrl_proto_sec_cfg::sec_params (*C++ member*), 177
esp_local_ctrl_proto_sec_cfg::version (*C++ member*), 177
esp_local_ctrl_proto_sec_cfg_t (*C++ type*), 179
esp_local_ctrl_proto_sec_t (*C++ type*), 179
esp_local_ctrl_remove_property (*C++ function*), 174
esp_local_ctrl_security1_params_t (*C++ type*), 179
esp_local_ctrl_security2_params_t (*C++ type*), 179
esp_local_ctrl_set_handler (*C++ function*), 174
esp_local_ctrl_start (*C++ function*), 174
esp_local_ctrl_stop (*C++ function*), 174
ESP_LOCAL_CTRL_TRANSPORT_BLE (*C macro*), 178
esp_local_ctrl_transport_config_ble_t (*C++ type*), 178
esp_local_ctrl_transport_config_httpd_t (*C++ type*), 178
esp_local_ctrl_transport_config_t (*C++ union*), 175
esp_local_ctrl_transport_config_t::ble (*C++ member*), 175
esp_local_ctrl_transport_config_t::httpd (*C++ member*), 175
ESP_LOCAL_CTRL_TRANSPORT_HTTPD (*C macro*), 178
esp_local_ctrl_transport_t (*C++ type*), 178
ESP_LOG_BUFFER_CHAR (*C macro*), 2078
ESP_LOG_BUFFER_CHAR_LEVEL (*C macro*), 2077
ESP_LOG_BUFFER_HEX (*C macro*), 2077
ESP_LOG_BUFFER_HEX_LEVEL (*C macro*), 2077
ESP_LOG_BUFFER_HEXDUMP (*C macro*), 2077
ESP_LOG_EARLY_IMPL (*C macro*), 2079
esp_log_early_timestamp (*C++ function*), 2076
ESP_LOG_LEVEL (*C macro*), 2079
esp_log_level_get (*C++ function*), 2076
ESP_LOG_LEVEL_LOCAL (*C macro*), 2079

esp_log_level_set (*C++ function*), 2075
esp_log_level_t (*C++ enum*), 2080
esp_log_level_t::ESP_LOG_DEBUG (*C++ enumerator*), 2081
esp_log_level_t::ESP_LOG_ERROR (*C++ enumerator*), 2080
esp_log_level_t::ESP_LOG_INFO (*C++ enumerator*), 2081
esp_log_level_t::ESP_LOG_NONE (*C++ enumerator*), 2080
esp_log_level_t::ESP_LOG_VERBOSE (*C++ enumerator*), 2081
esp_log_level_t::ESP_LOG_WARN (*C++ enumerator*), 2080
esp_log_set_vprintf (*C++ function*), 2076
esp_log_system_timestamp (*C++ function*), 2076
esp_log_timestamp (*C++ function*), 2076
esp_log_write (*C++ function*), 2077
esp_log_writev (*C++ function*), 2077
ESP_LOGD (*C macro*), 2079
ESP_LOGE (*C macro*), 2079
ESP_LOGI (*C macro*), 2079
ESP_LOGV (*C macro*), 2079
ESP_LOGW (*C macro*), 2079
esp_mac_type_t (*C++ enum*), 2088
esp_mac_type_t::ESP_MAC_BT (*C++ enumerator*), 2088
esp_mac_type_t::ESP_MAC_ETH (*C++ enumerator*), 2088
esp_mac_type_t::ESP_MAC_IEEE802154 (*C++ enumerator*), 2088
esp_mac_type_t::ESP_MAC_WIFI_SOFTAP (*C++ enumerator*), 2088
esp_mac_type_t::ESP_MAC_WIFI_STA (*C++ enumerator*), 2088
esp_mbo_update_non_pref_chan (*C++ function*), 940
esp_mesh_allow_root_conflicts (*C++ function*), 846
esp_mesh_available_txupQ_num (*C++ function*), 846
esp_mesh_connect (*C++ function*), 850
esp_mesh_deinit (*C++ function*), 836
esp_mesh_delete_group_id (*C++ function*), 847
esp_mesh_disable_ps (*C++ function*), 851
esp_mesh_disconnect (*C++ function*), 850
esp_mesh_enable_ps (*C++ function*), 851
esp_mesh_fix_root (*C++ function*), 848
esp_mesh_flush_scan_result (*C++ function*), 850
esp_mesh_flush_upstream_packets (*C++ function*), 850
esp_mesh_get_active_duty_cycle (*C++ function*), 852
esp_mesh_get_ap_assoc_expire (*C++ function*), 845
esp_mesh_get_ap_authmode (*C++ function*), 842
esp_mesh_get_ap_connections (*C++ function*), 843
esp_mesh_get_capacity_num (*C++ function*), 847
esp_mesh_get_config (*C++ function*), 840
esp_mesh_get_group_list (*C++ function*), 847
esp_mesh_get_group_num (*C++ function*), 847
esp_mesh_get_id (*C++ function*), 841
esp_mesh_get_ie_crypto_key (*C++ function*), 848
esp_mesh_get_layer (*C++ function*), 843
esp_mesh_get_max_layer (*C++ function*), 842
esp_mesh_get_network_duty_cycle (*C++ function*), 853
esp_mesh_get_non_mesh_connections (*C++ function*), 843
esp_mesh_get_parent_bssid (*C++ function*), 843
esp_mesh_get_root_healing_delay (*C++ function*), 848
esp_mesh_get_router (*C++ function*), 841
esp_mesh_get_router_bssid (*C++ function*), 851
esp_mesh_get_routing_table (*C++ function*), 845
esp_mesh_get_routing_table_size (*C++ function*), 845
esp_mesh_get_running_active_duty_cycle (*C++ function*), 853
esp_mesh_get_rx_pending (*C++ function*), 846
esp_mesh_get_self_organized (*C++ function*), 844
esp_mesh_get_subnet_nodes_list (*C++ function*), 850
esp_mesh_get_subnet_nodes_num (*C++ function*), 850
esp_mesh_get_topology (*C++ function*), 851
esp_mesh_get_total_node_num (*C++ function*), 845
esp_mesh_get_tsf_time (*C++ function*), 851
esp_mesh_get_tx_pending (*C++ function*), 846
esp_mesh_get_type (*C++ function*), 841
esp_mesh_get_vote_percentage (*C++ function*), 845
esp_mesh_get_xon_qsize (*C++ function*), 846
esp_mesh_init (*C++ function*), 836
esp_mesh_is_device_active (*C++ function*), 852
esp_mesh_is_my_group (*C++ function*), 847
esp_mesh_is_ps_enabled (*C++ function*), 852
esp_mesh_is_root (*C++ function*), 843
esp_mesh_is_root_conflicts_allowed (*C++ function*), 846
esp_mesh_is_root_fixed (*C++ function*), 848
esp_mesh_post_toDS_state (*C++ function*), 845

esp_mesh_ps_duty_signaling (C++ function), 854
esp_mesh_recv (C++ function), 839
esp_mesh_recv_toDS (C++ function), 839
esp_mesh_scan_get_ap_ie_len (C++ function), 849
esp_mesh_scan_get_ap_record (C++ function), 849
esp_mesh_send (C++ function), 837
esp_mesh_send_block_time (C++ function), 838
esp_mesh_set_active_duty_cycle (C++ function), 852
esp_mesh_set_ap_assoc_expire (C++ function), 845
esp_mesh_set_ap_authmode (C++ function), 842
esp_mesh_set_ap_connections (C++ function), 843
esp_mesh_set_ap_password (C++ function), 842
esp_mesh_set_capacity_num (C++ function), 847
esp_mesh_set_config (C++ function), 840
esp_mesh_set_group_id (C++ function), 847
esp_mesh_set_id (C++ function), 841
esp_mesh_set_ie_crypto_funcs (C++ function), 847
esp_mesh_set_ie_crypto_key (C++ function), 848
esp_mesh_set_max_layer (C++ function), 842
esp_mesh_set_network_duty_cycle (C++ function), 852
esp_mesh_set_parent (C++ function), 849
esp_mesh_set_root_healing_delay (C++ function), 848
esp_mesh_set_router (C++ function), 841
esp_mesh_set_self_organized (C++ function), 843
esp_mesh_set_topology (C++ function), 851
esp_mesh_set_type (C++ function), 841
esp_mesh_set_vote_percentage (C++ function), 844
esp_mesh_set_xon_qsize (C++ function), 846
esp_mesh_start (C++ function), 837
esp_mesh_stop (C++ function), 837
esp_mesh_switch_channel (C++ function), 850
esp_mesh_topology_t (C++ enum), 869
esp_mesh_topology_t::MESH_TOPO_CHAIN (C++ enumerator), 869
esp_mesh_topology_t::MESH_TOPO_TREE (C++ enumerator), 869
esp_mesh_waive_root (C++ function), 844
esp_mqtt_client_config_t (C++ struct), 132
esp_mqtt_client_config_t (C++ type), 138
esp_mqtt_client_config_t::broker (C++ member), 132
esp_mqtt_client_config_t::broker_t (C++ struct), 132
esp_mqtt_client_config_t::broker_t::address (C++ member), 132
esp_mqtt_client_config_t::broker_t::address_t (C++ struct), 133
esp_mqtt_client_config_t::broker_t::address_t::ho (C++ member), 133
esp_mqtt_client_config_t::broker_t::address_t::pa (C++ member), 133
esp_mqtt_client_config_t::broker_t::address_t::po (C++ member), 133
esp_mqtt_client_config_t::broker_t::address_t::tr (C++ member), 133
esp_mqtt_client_config_t::broker_t::address_t::ur (C++ member), 133
esp_mqtt_client_config_t::broker_t::verification (C++ member), 132
esp_mqtt_client_config_t::broker_t::verification_ (C++ struct), 133
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 134
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 133
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 133
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 133
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 133
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 134
esp_mqtt_client_config_t::broker_t::verification_ (C++ member), 133
esp_mqtt_client_config_t::buffer (C++ member), 132
esp_mqtt_client_config_t::buffer_t (C++ struct), 134
esp_mqtt_client_config_t::buffer_t::out_size (C++ member), 134
esp_mqtt_client_config_t::buffer_t::size (C++ member), 134
esp_mqtt_client_config_t::credentials (C++ member), 132
esp_mqtt_client_config_t::credentials_t (C++ struct), 134
esp_mqtt_client_config_t::credentials_t::authenti (C++ member), 134
esp_mqtt_client_config_t::credentials_t::authenti (C++ struct), 134
esp_mqtt_client_config_t::credentials_t::authenti (C++ member), 135
esp_mqtt_client_config_t::credentials_t::authenti (C++ member), 135
esp_mqtt_client_config_t::credentials_t::authenti (C++ member), 135
esp_mqtt_client_config_t::credentials_t::authenti (C++ member), 135
esp_mqtt_client_config_t::credentials_t::authenti

(C++ *member*), 135
esp_mqtt_client_config_t::credentials_t::authentication_t::certificate (C++ *member*), 135
esp_mqtt_client_config_t::credentials_t::authentication_t::key_password (C++ *member*), 135
esp_mqtt_client_config_t::credentials_t::authentication_t::key_pem_vault_handle (C++ *member*), 135
esp_mqtt_client_config_t::credentials_t::authentication_t::password (C++ *member*), 135
esp_mqtt_client_config_t::credentials_t::client_id (C++ *member*), 134
esp_mqtt_client_config_t::credentials_t::set_null_client_id (C++ *member*), 134
esp_mqtt_client_config_t::credentials_t::username (C++ *member*), 134
esp_mqtt_client_config_t::network (C++ *member*), 132
esp_mqtt_client_config_t::network_t (C++ *struct*), 135
esp_mqtt_client_config_t::network_t::disable_auto_reconnect (C++ *member*), 135
esp_mqtt_client_config_t::network_t::reconnect_timeout_ms (C++ *member*), 135
esp_mqtt_client_config_t::network_t::refresh_connection_after_ms (C++ *member*), 135
esp_mqtt_client_config_t::network_t::timeout_ms (C++ *member*), 135
esp_mqtt_client_config_t::session (C++ *member*), 132
esp_mqtt_client_config_t::session_t (C++ *struct*), 136
esp_mqtt_client_config_t::session_t::disable_clean_session (C++ *member*), 136
esp_mqtt_client_config_t::session_t::disable_keepalive (C++ *member*), 136
esp_mqtt_client_config_t::session_t::keepalive (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will_t (C++ *struct*), 136
esp_mqtt_client_config_t::session_t::last_will_t::msg (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will_t::msg_len (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will_t::qos (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will_t::retain (C++ *member*), 136
esp_mqtt_client_config_t::session_t::last_will_t::topic (C++ *member*), 136
esp_mqtt_client_config_t::session_t::message_retransmit_timeout (C++ *member*), 136
esp_mqtt_client_config_t::session_t::protocol_ver (C++ *member*), 136
esp_mqtt_client_config_t::task (C++ *member*), 132
esp_mqtt_client_config_t::task_t (C++ *struct*), 136
esp_mqtt_client_config_t::task_t::priority (C++ *member*), 137
esp_mqtt_client_config_t::task_t::stack_size (C++ *member*), 137
esp_mqtt_client_destroy (C++ *function*), 129
esp_mqtt_client_disconnect (C++ *function*), 128
esp_mqtt_client_enqueue (C++ *function*), 129
esp_mqtt_client_get_outbox_size (C++ *function*), 130
esp_mqtt_client_handle_t (C++ *type*), 137
esp_mqtt_client_init (C++ *function*), 127
esp_mqtt_client_publish (C++ *function*), 129
esp_mqtt_client_reconnect (C++ *function*), 128
esp_mqtt_client_register_event (C++ *function*), 130
esp_mqtt_client_set_uri (C++ *function*), 128
esp_mqtt_client_start (C++ *function*), 128
esp_mqtt_client_stop (C++ *function*), 128
esp_mqtt_client_subscribe (C++ *function*), 128
esp_mqtt_client_unsubscribe (C++ *function*), 128
esp_mqtt_connect_return_code_t (C++ *enum*), 139
esp_mqtt_connect_return_code_t (C++ *type*), 137
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_A (C++ *enumerator*), 139
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_R (C++ *enumerator*), 139
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_R (C++ *enumerator*), 139
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_R (C++ *enumerator*), 139
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_R (C++ *enumerator*), 139
esp_mqtt_connect_return_code_t::MQTT_CONNECTION_R (C++ *enumerator*), 139
esp_mqtt_error_codes (C++ *struct*), 130
esp_mqtt_error_codes::connect_return_code (C++ *member*), 131
esp_mqtt_error_codes::error_type (C++ *member*), 130
esp_mqtt_error_codes::esp_tls_cert_verify_flags (C++ *member*), 130
esp_mqtt_error_codes::esp_tls_last_esp_err (C++ *member*), 130
esp_mqtt_error_codes::esp_tls_stack_err (C++ *member*), 130
esp_mqtt_error_codes::esp_transport_sock_errno (C++ *member*), 131
esp_mqtt_error_codes_t (C++ *type*), 137
esp_mqtt_error_type_t (C++ *enum*), 139
esp_mqtt_error_type_t (C++ *type*), 137
esp_mqtt_error_type_t::MQTT_ERROR_TYPE_CONNECTION

(C++ *enumerator*), 140
esp_mqtt_error_type_t::MQTT_ERROR_TYPE_NONE (C++ *enumerator*), 140
esp_mqtt_error_type_t::MQTT_ERROR_TYPE_TCP_TRANSPORT (C++ *enumerator*), 140
esp_mqtt_event_handle_t (C++ *type*), 138
esp_mqtt_event_id_t (C++ *enum*), 138
esp_mqtt_event_id_t (C++ *type*), 137
esp_mqtt_event_id_t::MQTT_EVENT_ANY (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_BEFORE_CONNECT (C++ *enumerator*), 139
esp_mqtt_event_id_t::MQTT_EVENT_CONNECTED (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_DATA (C++ *enumerator*), 139
esp_mqtt_event_id_t::MQTT_EVENT_DELETED (C++ *enumerator*), 139
esp_mqtt_event_id_t::MQTT_EVENT_DISCONNECTED (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_ERROR (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_PUBLISHED (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_SUBSCRIBED (C++ *enumerator*), 138
esp_mqtt_event_id_t::MQTT_EVENT_UNSUBSCRIBED (C++ *enumerator*), 138
esp_mqtt_event_t (C++ *struct*), 131
esp_mqtt_event_t (C++ *type*), 138
esp_mqtt_event_t::client (C++ *member*), 131
esp_mqtt_event_t::current_data_offset (C++ *member*), 131
esp_mqtt_event_t::data (C++ *member*), 131
esp_mqtt_event_t::data_len (C++ *member*), 131
esp_mqtt_event_t::dup (C++ *member*), 132
esp_mqtt_event_t::error_handle (C++ *member*), 131
esp_mqtt_event_t::event_id (C++ *member*), 131
esp_mqtt_event_t::msg_id (C++ *member*), 131
esp_mqtt_event_t::protocol_ver (C++ *member*), 132
esp_mqtt_event_t::qos (C++ *member*), 132
esp_mqtt_event_t::retain (C++ *member*), 131
esp_mqtt_event_t::session_present (C++ *member*), 131
esp_mqtt_event_t::topic (C++ *member*), 131
esp_mqtt_event_t::topic_len (C++ *member*), 131
esp_mqtt_event_t::total_data_len (C++ *member*), 131
esp_mqtt_protocol_ver_t (C++ *enum*), 140
esp_mqtt_protocol_ver_t (C++ *type*), 137
esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_UNDEFINED (C++ *enumerator*), 140
esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_3_1 (C++ *enumerator*), 140
esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_3_1_1 (C++ *enumerator*), 140
esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_5 (C++ *enumerator*), 140
esp_mqtt_set_config (C++ *function*), 129
esp_mqtt_transport_t (C++ *enum*), 140
esp_mqtt_transport_t (C++ *type*), 137
esp_mqtt_transport_t::MQTT_TRANSPORT_OVER_SSL (C++ *enumerator*), 140
esp_mqtt_transport_t::MQTT_TRANSPORT_OVER_TCP (C++ *enumerator*), 140
esp_mqtt_transport_t::MQTT_TRANSPORT_OVER_WS (C++ *enumerator*), 140
esp_mqtt_transport_t::MQTT_TRANSPORT_OVER_WSS (C++ *enumerator*), 140
esp_mqtt_transport_t::MQTT_TRANSPORT_UNKNOWN (C++ *enumerator*), 140
esp_netif_action_add_ip6_address (C++ *function*), 995
esp_netif_action_connected (C++ *function*), 994
esp_netif_action_disconnected (C++ *function*), 994
esp_netif_action_got_ip (C++ *function*), 995
esp_netif_action_join_ip6_multicast_group (C++ *function*), 995
esp_netif_action_leave_ip6_multicast_group (C++ *function*), 995
esp_netif_action_remove_ip6_address (C++ *function*), 996
esp_netif_action_start (C++ *function*), 994
esp_netif_action_stop (C++ *function*), 994
esp_netif_attach (C++ *function*), 993
esp_netif_attach_wifi_ap (C++ *function*), 1017
esp_netif_attach_wifi_station (C++ *function*), 1017
ESP_NETIF_BR_DROP (C *macro*), 1010
ESP_NETIF_BR_FDW_CPU (C *macro*), 1010
ESP_NETIF_BR_FLOOD (C *macro*), 1010
esp_netif_callback_fn (C++ *type*), 1005
esp_netif_config (C++ *struct*), 1008
esp_netif_config::base (C++ *member*), 1008
esp_netif_config::driver (C++ *member*), 1008
esp_netif_config::stack (C++ *member*), 1009
esp_netif_config_t (C++ *type*), 1010
esp_netif_create_default_wifi_ap (C++ *function*), 1018
esp_netif_create_default_wifi_mesh_netifs (C++ *function*), 1018
esp_netif_create_default_wifi_sta (C++ *function*), 1018

esp_netif_create_ip6_linklocal (C++ function), 1002
esp_netif_create_wifi (C++ function), 1018
ESP_NETIF_DEFAULT_OPENTHREAD (C macro), 986
esp_netif_deinit (C++ function), 993
esp_netif_destroy (C++ function), 993
esp_netif_destroy_default_wifi (C++ function), 1018
esp_netif_dhcp_option_id_t (C++ enum), 1011
esp_netif_dhcp_option_id_t::ESP_NETIF_DOMAIN_NAME_SERVER (C++ enumerator), 1011
esp_netif_dhcp_option_id_t::ESP_NETIF_IP_ADDRESS_LEASE_TIME (C++ enumerator), 1012
esp_netif_dhcp_option_id_t::ESP_NETIF_IP_REQUEST_RETRY_TIME (C++ enumerator), 1012
esp_netif_dhcp_option_id_t::ESP_NETIF_REQUESTED_IP_ADDRESS (C++ enumerator), 1012
esp_netif_dhcp_option_id_t::ESP_NETIF_ROUTER_SOLICITATION_ADDRESS (C++ enumerator), 1011
esp_netif_dhcp_option_id_t::ESP_NETIF_SUBNET_MASK (C++ enumerator), 1011
esp_netif_dhcp_option_id_t::ESP_NETIF_VENDOR_CLASS_IDENTIFIER (C++ enumerator), 1012
esp_netif_dhcp_option_id_t::ESP_NETIF_VENDOR_SPECIFIC_INFO (C++ enumerator), 1012
esp_netif_dhcp_option_mode_t (C++ enum), 1011
esp_netif_dhcp_option_mode_t::ESP_NETIF_OP_GET (C++ enumerator), 1011
esp_netif_dhcp_option_mode_t::ESP_NETIF_OP_MAX (C++ enumerator), 1011
esp_netif_dhcp_option_mode_t::ESP_NETIF_OP_SET (C++ enumerator), 1011
esp_netif_dhcp_option_mode_t::ESP_NETIF_OP_START (C++ enumerator), 1011
esp_netif_dhcp_status_t (C++ enum), 1011
esp_netif_dhcp_status_t::ESP_NETIF_DHCP_INIT (C++ enumerator), 1011
esp_netif_dhcp_status_t::ESP_NETIF_DHCP_STARTED (C++ enumerator), 1011
esp_netif_dhcp_status_t::ESP_NETIF_DHCP_STATUS_MAX (C++ enumerator), 1011
esp_netif_dhcp_status_t::ESP_NETIF_DHCP_STOPPED (C++ enumerator), 1011
esp_netif_dhcpc_get_status (C++ function), 1000
esp_netif_dhcpc_option (C++ function), 999
esp_netif_dhcpc_start (C++ function), 1000
esp_netif_dhcpc_stop (C++ function), 1000
esp_netif_dhcps_get_clients_by_mac (C++ function), 1001
esp_netif_dhcps_get_status (C++ function), 1000
esp_netif_dhcps_option (C++ function), 999
esp_netif_dhcps_start (C++ function), 1000
esp_netif_dhcps_stop (C++ function), 1000
esp_netif_dns_info_t (C++ struct), 1005
esp_netif_dns_info_t::ip (C++ member), 1005
esp_netif_dns_type_t (C++ enum), 1010
esp_netif_dns_type_t::ESP_NETIF_DNS_BACKUP (C++ enumerator), 1010
esp_netif_dns_type_t::ESP_NETIF_DNS_FALLBACK (C++ enumerator), 1011
esp_netif_dns_type_t::ESP_NETIF_DNS_MAIN (C++ enumerator), 1010
esp_netif_dns_type_t::ESP_NETIF_DNS_MAX (C++ enumerator), 1011
esp_netif_driver_base_s (C++ struct), 1008
esp_netif_driver_base_s::netif (C++ member), 1008
esp_netif_driver_base_s::post_attach (C++ member), 1008
esp_netif_driver_base_t (C++ type), 1010
esp_netif_driver_ifconfig (C++ struct), 1008
esp_netif_driver_ifconfig::driver_free_rx_buffer (C++ member), 1008
esp_netif_driver_ifconfig::handle (C++ member), 1008
esp_netif_driver_ifconfig::transmit (C++ member), 1008
esp_netif_driver_ifconfig::transmit_wrap (C++ member), 1008
esp_netif_driver_ifconfig_t (C++ type), 1010
esp_netif_find_if (C++ function), 1004
esp_netif_find_predicate_t (C++ type), 1005
esp_netif_flags (C++ enum), 1012
esp_netif_flags::ESP_NETIF_DHCP_CLIENT (C++ enumerator), 1012
esp_netif_flags::ESP_NETIF_DHCP_SERVER (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_AUTOUP (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_EVENT_IP_MODIFIED (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_GARP (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_IS_BRIDGE (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_IS_PPP (C++ enumerator), 1013
esp_netif_flags::ESP_NETIF_FLAG_MLDV6_REPORT (C++ enumerator), 1013
esp_netif_flags_t (C++ type), 1010
esp_netif_free_rx_buffer (C++ function), 1021
esp_netif_get_all_ip6 (C++ function), 1002
esp_netif_get_desc (C++ function), 1003
esp_netif_get_dns_info (C++ function), 1001
esp_netif_get_event_id (C++ function), 1004
esp_netif_get_flags (C++ function), 1003

esp_netif_get_handle_from_ifkey (C++ function), 1003
esp_netif_get_handle_from_netif_impl (C++ function), 1020
esp_netif_get_hostname (C++ function), 997
esp_netif_get_ifkey (C++ function), 1003
esp_netif_get_io_driver (C++ function), 1003
esp_netif_get_ip6_global (C++ function), 1002
esp_netif_get_ip6_linklocal (C++ function), 1002
esp_netif_get_ip_info (C++ function), 997
esp_netif_get_mac (C++ function), 997
esp_netif_get_netif_impl (C++ function), 1020
esp_netif_get_netif_impl_index (C++ function), 999
esp_netif_get_netif_impl_name (C++ function), 999
esp_netif_get_nr_of_ifs (C++ function), 1004
esp_netif_get_old_ip_info (C++ function), 998
esp_netif_get_route_prio (C++ function), 1004
esp_netif_htonl (C macro), 1014
esp_netif_inherent_config (C++ struct), 1007
esp_netif_inherent_config::bridge_info (C++ member), 1008
esp_netif_inherent_config::flags (C++ member), 1007
esp_netif_inherent_config::get_ip_event (C++ member), 1007
esp_netif_inherent_config::if_desc (C++ member), 1007
esp_netif_inherent_config::if_key (C++ member), 1007
esp_netif_inherent_config::ip_info (C++ member), 1007
esp_netif_inherent_config::lost_ip_event (C++ member), 1007
esp_netif_inherent_config::mac (C++ member), 1007
esp_netif_inherent_config::route_prio (C++ member), 1008
esp_netif_inherent_config_t (C++ type), 1010
ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD (C macro), 986
esp_netif_init (C++ function), 992
esp_netif_iodriver_handle (C++ type), 1010
esp_netif_ip4_makeu32 (C macro), 1014
esp_netif_ip6_get_addr_type (C++ function), 1013
esp_netif_ip6_info_t (C++ struct), 1005
esp_netif_ip6_info_t::ip (C++ member), 1006
esp_netif_ip_addr_copy (C++ function), 1013
esp_netif_ip_event_type (C++ enum), 1013
esp_netif_ip_event_type::ESP_NETIF_IP_EVENT_GOT_I (C++ enumerator), 1013
esp_netif_ip_event_type::ESP_NETIF_IP_EVENT_LOST_ (C++ enumerator), 1013
esp_netif_ip_event_type_t (C++ type), 1010
esp_netif_ip_info_t (C++ struct), 1005
esp_netif_ip_info_t::gw (C++ member), 1005
esp_netif_ip_info_t::ip (C++ member), 1005
esp_netif_ip_info_t::netmask (C++ member), 1005
esp_netif_is_netif_up (C++ function), 997
esp_netif_join_ip6_multicast_group (C++ function), 996
esp_netif_leave_ip6_multicast_group (C++ function), 996
esp_netif_netstack_buf_free (C++ function), 1005
esp_netif_netstack_buf_ref (C++ function), 1004
esp_netif_netstack_config_t (C++ type), 1010
esp_netif_new (C++ function), 993
esp_netif_next (C++ function), 1004
esp_netif_next_unsafe (C++ function), 1004
esp_netif_pair_mac_ip_t (C++ struct), 1009
esp_netif_pair_mac_ip_t::ip (C++ member), 1009
esp_netif_pair_mac_ip_t::mac (C++ member), 1009
esp_netif_receive (C++ function), 993
esp_netif_receive_t (C++ type), 1010
esp_netif_set_default_netif (C++ function), 996
esp_netif_set_dns_info (C++ function), 1001
esp_netif_set_driver_config (C++ function), 993
esp_netif_set_hostname (C++ function), 997
esp_netif_set_ip4_addr (C++ function), 1002
esp_netif_set_ip_info (C++ function), 998
esp_netif_set_link_speed (C++ function), 1020
esp_netif_set_mac (C++ function), 996
esp_netif_set_old_ip_info (C++ function), 998
esp_netif_str_to_ip4 (C++ function), 1003
esp_netif_str_to_ip6 (C++ function), 1003
esp_netif_t (C++ type), 1010
esp_netif_tcpip_exec (C++ function), 1005
esp_netif_transmit (C++ function), 1021
esp_netif_transmit_wrap (C++ function), 1021
esp_nimble_hci_deinit (C++ function), 816
esp_nimble_hci_init (C++ function), 815

esp_now_add_peer (*C++ function*), 826
esp_now_deinit (*C++ function*), 825
esp_now_del_peer (*C++ function*), 826
ESP_NOW_ETH_ALEN (*C macro*), 830
esp_now_fetch_peer (*C++ function*), 827
esp_now_get_peer (*C++ function*), 827
esp_now_get_peer_num (*C++ function*), 827
esp_now_get_version (*C++ function*), 825
esp_now_init (*C++ function*), 825
esp_now_is_peer_exist (*C++ function*), 827
ESP_NOW_KEY_LEN (*C macro*), 830
ESP_NOW_MAX_DATA_LEN (*C macro*), 830
ESP_NOW_MAX_ENCRYPT_PEER_NUM (*C macro*), 830
ESP_NOW_MAX_TOTAL_PEER_NUM (*C macro*), 830
esp_now_mod_peer (*C++ function*), 826
esp_now_peer_info (*C++ struct*), 828
esp_now_peer_info::channel (*C++ member*), 828
esp_now_peer_info::encrypt (*C++ member*), 828
esp_now_peer_info::ifidx (*C++ member*), 828
esp_now_peer_info::lmk (*C++ member*), 828
esp_now_peer_info::peer_addr (*C++ member*), 828
esp_now_peer_info::priv (*C++ member*), 828
esp_now_peer_info_t (*C++ type*), 830
esp_now_peer_num (*C++ struct*), 828
esp_now_peer_num::encrypt_num (*C++ member*), 829
esp_now_peer_num::total_num (*C++ member*), 829
esp_now_peer_num_t (*C++ type*), 830
esp_now_recv_cb_t (*C++ type*), 830
esp_now_recv_info (*C++ struct*), 829
esp_now_recv_info::des_addr (*C++ member*), 829
esp_now_recv_info::rx_ctrl (*C++ member*), 829
esp_now_recv_info::src_addr (*C++ member*), 829
esp_now_recv_info_t (*C++ type*), 830
esp_now_register_recv_cb (*C++ function*), 825
esp_now_register_send_cb (*C++ function*), 825
esp_now_send (*C++ function*), 826
esp_now_send_cb_t (*C++ type*), 830
esp_now_send_status_t (*C++ enum*), 830
esp_now_send_status_t::ESP_NOW_SEND_FAIL (*C++ enumerator*), 831
esp_now_send_status_t::ESP_NOW_SEND_SUCCESS (*C++ enumerator*), 830
esp_now_set_pmk (*C++ function*), 827
esp_now_set_wake_window (*C++ function*), 828
esp_now_unregister_recv_cb (*C++ function*), 825
esp_now_unregister_send_cb (*C++ function*), 826
ESP_OK (*C macro*), 1871
ESP_OK_EFUSE_CNT (*C macro*), 1869
esp_openthread_border_router_deinit (*C++ function*), 986
esp_openthread_border_router_init (*C++ function*), 986
esp_openthread_deinit (*C++ function*), 981
esp_openthread_event_t (*C++ enum*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_GOT_IP6 (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_IF_DOWN (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_IF_UP (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_LOST_IP6 (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_MULTICAS (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_MULTICAS (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_START (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_STOP (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_TREL_ADD (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_TREL_MUL (*C++ enumerator*), 984
esp_openthread_event_t::OPENTHREAD_EVENT_TREL_REM (*C++ enumerator*), 984
esp_openthread_get_backbone_netif (*C++ function*), 986
esp_openthread_get_instance (*C++ function*), 981
esp_openthread_get_netif (*C++ function*), 986
esp_openthread_host_connection_config_t (*C++ struct*), 983
esp_openthread_host_connection_config_t::host_con (*C++ member*), 983
esp_openthread_host_connection_config_t::host_uar (*C++ member*), 983
esp_openthread_host_connection_mode_t (*C++ enum*), 985
esp_openthread_host_connection_mode_t::HOST_CONNE (*C++ enumerator*), 985
esp_openthread_host_connection_mode_t::HOST_CONNE (*C++ enumerator*), 985
esp_openthread_host_connection_mode_t::HOST_CONNE (*C++ enumerator*), 985
esp_openthread_init (*C++ function*), 981
esp_openthread_launch_mainloop (*C++ function*), 981
esp_openthread_lock_acquire (*C++ function*), 985
esp_openthread_lock_deinit (*C++ function*),

985
esp_openthread_lock_init (C++ function), 985
esp_openthread_lock_release (C++ function), 985
esp_openthread_mainloop_context_t (C++ struct), 982
esp_openthread_mainloop_context_t::error_fds (C++ member), 982
esp_openthread_mainloop_context_t::max_fd (C++ member), 982
esp_openthread_mainloop_context_t::read_fds (C++ member), 982
esp_openthread_mainloop_context_t::timeout (C++ member), 982
esp_openthread_mainloop_context_t::write_fds (C++ member), 982
esp_openthread_netif_glue_deinit (C++ function), 986
esp_openthread_netif_glue_init (C++ function), 986
esp_openthread_platform_config_t (C++ struct), 983
esp_openthread_platform_config_t::host_config (C++ member), 983
esp_openthread_platform_config_t::port_config (C++ member), 983
esp_openthread_platform_config_t::radio_config (C++ member), 983
esp_openthread_port_config_t (C++ struct), 983
esp_openthread_port_config_t::netif_queue_size (C++ member), 983
esp_openthread_port_config_t::storage_partition_name (C++ member), 983
esp_openthread_port_config_t::task_queue_size (C++ member), 983
esp_openthread_radio_config_t (C++ struct), 982
esp_openthread_radio_config_t::radio_mode (C++ member), 983
esp_openthread_radio_config_t::radio_uart_config (C++ member), 983
esp_openthread_radio_mode_t (C++ enum), 984
esp_openthread_radio_mode_t::RADIO_MODE_NATIVE (C++ enumerator), 984
esp_openthread_radio_mode_t::RADIO_MODE_SPI_RCP (C++ enumerator), 985
esp_openthread_radio_mode_t::RADIO_MODE_UART_RCP (C++ enumerator), 984
esp_openthread_rcp_deinit (C++ function), 987
esp_openthread_rcp_failure_handler (C++ type), 984
esp_openthread_register_rcp_failure_handler (C++ function), 987
esp_openthread_set_backbone_netif (C++ function), 986
esp_openthread_uart_config_t (C++ struct), 982
esp_openthread_uart_config_t::port (C++ member), 982
esp_openthread_uart_config_t::rx_pin (C++ member), 982
esp_openthread_uart_config_t::tx_pin (C++ member), 982
esp_openthread_uart_config_t::uart_config (C++ member), 982
esp_ota_abort (C++ function), 2103
esp_ota_begin (C++ function), 2102
esp_ota_check_rollback_is_possible (C++ function), 2106
esp_ota_end (C++ function), 2103
esp_ota_erase_last_boot_app_partition (C++ function), 2105
esp_ota_get_app_description (C++ function), 2101
esp_ota_get_app_elf_sha256 (C++ function), 2101
esp_ota_get_app_partition_count (C++ function), 2105
esp_ota_get_boot_partition (C++ function), 2104
esp_ota_get_last_invalid_partition (C++ function), 2105
esp_ota_get_next_update_partition (C++ function), 2104
esp_ota_get_partition_description (C++ function), 2104
esp_ota_get_running_partition (C++ function), 2104
esp_ota_get_state_partition (C++ function), 2105
esp_ota_handle_t (C++ type), 2106
esp_ota_mark_app_invalid_rollback_and_reboot (C++ function), 2105
esp_ota_mark_app_valid_cancel_rollback (C++ function), 2105
esp_ota_set_boot_partition (C++ function), 2103
esp_ota_write (C++ function), 2102
esp_ota_write_with_offset (C++ function), 2103
esp_partition_check_identity (C++ function), 1796
esp_partition_deregister_external (C++ function), 1797
esp_partition_erase_range (C++ function), 1795
esp_partition_find (C++ function), 1793
esp_partition_find_first (C++ function), 1793
esp_partition_get (C++ function), 1793
esp_partition_get_sha256 (C++ function), 1796

esp_partition_iterator_release (*C++ function*), 1793
esp_partition_iterator_t (*C++ type*), 1798
esp_partition_mmap (*C++ function*), 1795
esp_partition_mmap_handle_t (*C++ type*), 1798
esp_partition_mmap_memory_t (*C++ enum*), 1798
esp_partition_mmap_memory_t::ESP_PARTITION_MMAP_DATA (*C++ enumerator*), 1798
esp_partition_mmap_memory_t::ESP_PARTITION_MMAP_INST (*C++ enumerator*), 1798
esp_partition_munmap (*C++ function*), 1796
esp_partition_next (*C++ function*), 1793
esp_partition_read (*C++ function*), 1794
esp_partition_read_raw (*C++ function*), 1794
esp_partition_register_external (*C++ function*), 1797
ESP_PARTITION_SUBTYPE_OTA (*C macro*), 1798
esp_partition_subtype_t (*C++ enum*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_ANY (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_FACTORY (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_0 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_1 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_10 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_11 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_12 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_13 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_14 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_15 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_2 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_3 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_4 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_5 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_6 (*C++ enumerator*), 1799
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_7 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_8 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_9 (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_APP_OTA_MAX (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_COREDUMP (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_FAT (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_NVS (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_OTA (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_PHY (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_SPIFFS (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_UNDEFINED (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_DATA_WIFI (*C++ enumerator*), 1800
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_OTA_FLAG (*C++ enumerator*), 1801
esp_partition_subtype_t::ESP_PARTITION_SUBTYPE_OTA_OTA_DATA (*C++ enumerator*), 1801
esp_partition_t (*C++ struct*), 1797
esp_partition_t::address (*C++ member*), 1798
esp_partition_t::encrypted (*C++ member*), 1798
esp_partition_t::erase_size (*C++ member*), 1798
esp_partition_t::flash_chip (*C++ member*), 1797
esp_partition_t::label (*C++ member*), 1798
esp_partition_t::size (*C++ member*), 1798
esp_partition_t::subtype (*C++ member*), 1798
esp_partition_t::type (*C++ member*), 1797
esp_partition_type_t (*C++ enum*), 1798
esp_partition_type_t::ESP_PARTITION_TYPE_ANY (*C++ enumerator*), 1799
esp_partition_type_t::ESP_PARTITION_TYPE_APP (*C++ enumerator*), 1799
esp_partition_type_t::ESP_PARTITION_TYPE_DATA (*C++ enumerator*), 1799
esp_partition_verify (*C++ function*), 1793
esp_partition_write (*C++ function*), 1794
esp_partition_write_raw (*C++ function*), 1795
esp_phy_calibration_data_t (*C++ struct*), 2440
esp_phy_calibration_data_t::mac (*C++ member*), 2440
esp_phy_calibration_data_t::opaque (*C++ member*), 2440
esp_phy_calibration_data_t::version

(C++ *member*), 2440
esp_phy_calibration_mode_t (C++ *enum*), 2440
esp_phy_calibration_mode_t::PHY_RF_CAL_FULL (C++ *enumerator*), 2440
esp_phy_calibration_mode_t::PHY_RF_CAL_NONE (C++ *enumerator*), 2440
esp_phy_calibration_mode_t::PHY_RF_CAL_PARTIAL (C++ *enumerator*), 2440
esp_phy_common_clock_disable (C++ *function*), 2439
esp_phy_common_clock_enable (C++ *function*), 2439
esp_phy_disable (C++ *function*), 2439
esp_phy_enable (C++ *function*), 2439
esp_phy_erase_cal_data_in_nvs (C++ *function*), 2438
esp_phy_get_init_data (C++ *function*), 2438
esp_phy_init_data_t (C++ *struct*), 2439
esp_phy_init_data_t::params (C++ *member*), 2440
esp_phy_is_initialized (C++ *function*), 2438
esp_phy_load_cal_and_init (C++ *function*), 2439
esp_phy_load_cal_data_from_nvs (C++ *function*), 2438
esp_phy_modem_deinit (C++ *function*), 2439
esp_phy_modem_init (C++ *function*), 2439
esp_phy_release_init_data (C++ *function*), 2438
esp_phy_rf_get_on_ts (C++ *function*), 2439
esp_phy_store_cal_data_to_nvs (C++ *function*), 2438
esp_phy_update_country_info (C++ *function*), 2439
esp_ping_callbacks_t (C++ *struct*), 232
esp_ping_callbacks_t::cb_args (C++ *member*), 232
esp_ping_callbacks_t::on_ping_end (C++ *member*), 232
esp_ping_callbacks_t::on_ping_success (C++ *member*), 232
esp_ping_callbacks_t::on_ping_timeout (C++ *member*), 232
esp_ping_config_t (C++ *struct*), 232
esp_ping_config_t::count (C++ *member*), 232
esp_ping_config_t::data_size (C++ *member*), 232
esp_ping_config_t::interface (C++ *member*), 233
esp_ping_config_t::interval_ms (C++ *member*), 232
esp_ping_config_t::target_addr (C++ *member*), 232
esp_ping_config_t::task_prio (C++ *member*), 233
esp_ping_config_t::task_stack_size (C++ *member*), 233
esp_ping_config_t::timeout_ms (C++ *member*), 232
esp_ping_config_t::tos (C++ *member*), 232
esp_ping_config_t::ttl (C++ *member*), 232
ESP_PING_COUNT_INFINITE (C *macro*), 233
ESP_PING_DEFAULT_CONFIG (C *macro*), 233
esp_ping_delete_session (C++ *function*), 231
esp_ping_get_profile (C++ *function*), 231
esp_ping_handle_t (C++ *type*), 233
esp_ping_new_session (C++ *function*), 231
esp_ping_profile_t (C++ *enum*), 233
esp_ping_profile_t::ESP_PING_PROF_DURATION (C++ *enumerator*), 234
esp_ping_profile_t::ESP_PING_PROF_IPADDR (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_REPLY (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_REQUEST (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_SEQNO (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_SIZE (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_TIMEGAP (C++ *enumerator*), 234
esp_ping_profile_t::ESP_PING_PROF_TOS (C++ *enumerator*), 233
esp_ping_profile_t::ESP_PING_PROF_TTL (C++ *enumerator*), 233
esp_ping_start (C++ *function*), 231
esp_ping_stop (C++ *function*), 231
esp_pm_config_esp32_t (C++ *struct*), 2115
esp_pm_config_esp32_t::light_sleep_enable (C++ *member*), 2116
esp_pm_config_esp32_t::max_freq_mhz (C++ *member*), 2115
esp_pm_config_esp32_t::min_freq_mhz (C++ *member*), 2115
esp_pm_configure (C++ *function*), 2113
esp_pm_dump_locks (C++ *function*), 2115
esp_pm_get_configuration (C++ *function*), 2113
esp_pm_lock_acquire (C++ *function*), 2114
esp_pm_lock_create (C++ *function*), 2113
esp_pm_lock_delete (C++ *function*), 2114
esp_pm_lock_handle_t (C++ *type*), 2115
esp_pm_lock_release (C++ *function*), 2114
esp_pm_lock_type_t (C++ *enum*), 2115
esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX (C++ *enumerator*), 2115
esp_pm_lock_type_t::ESP_PM_CPU_FREQ_MAX (C++ *enumerator*), 2115
esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP (C++ *enumerator*), 2115
esp_power_level_t (C++ *enum*), 530
esp_power_level_t::ESP_PWR_LVL_N0 (C++ *enumerator*), 531

esp_power_level_t::ESP_PWR_LVL_N11 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_N12 (C++ enumerator), 530
esp_power_level_t::ESP_PWR_LVL_N14 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_N2 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_N3 (C++ enumerator), 530
esp_power_level_t::ESP_PWR_LVL_N5 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_N6 (C++ enumerator), 530
esp_power_level_t::ESP_PWR_LVL_N8 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_N9 (C++ enumerator), 530
esp_power_level_t::ESP_PWR_LVL_P1 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_P3 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_P4 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_P6 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_P7 (C++ enumerator), 531
esp_power_level_t::ESP_PWR_LVL_P9 (C++ enumerator), 531
esp_pthread_cfg_t (C++ struct), 2119
esp_pthread_cfg_t::inherit_cfg (C++ member), 2120
esp_pthread_cfg_t::pin_to_core (C++ member), 2120
esp_pthread_cfg_t::prio (C++ member), 2120
esp_pthread_cfg_t::stack_size (C++ member), 2120
esp_pthread_cfg_t::thread_name (C++ member), 2120
esp_pthread_get_cfg (C++ function), 2119
esp_pthread_get_default_config (C++ function), 2119
esp_pthread_init (C++ function), 2119
esp_pthread_set_cfg (C++ function), 2119
esp_random (C++ function), 2121
esp_read_mac (C++ function), 2087
esp_register_freertos_idle_hook (C++ function), 2029
esp_register_freertos_idle_hook_for_cpu (C++ function), 2029
esp_register_freertos_tick_hook (C++ function), 2030
esp_register_freertos_tick_hook_for_cpu (C++ function), 2029
esp_register_shutdown_handler (C++ function), 2084
esp_reset_reason (C++ function), 2084
esp_reset_reason_t (C++ enum), 2085
esp_reset_reason_t::ESP_RST_BROWNOUT (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_DEEPSLEEP (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_EXT (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_INT_WDT (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_PANIC (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_POWERON (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_SDIO (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_SW (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_TASK_WDT (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_UNKNOWN (C++ enumerator), 2085
esp_reset_reason_t::ESP_RST_WDT (C++ enumerator), 2085
esp_restart (C++ function), 2084
ESP_RETURN_ON_ERROR (C macro), 1870
ESP_RETURN_ON_ERROR_ISR (C macro), 1870
ESP_RETURN_ON_FALSE (C macro), 1870
ESP_RETURN_ON_FALSE_ISR (C macro), 1870
esp_rom_delay_us (C++ function), 2061
esp_rom_get_cpu_ticks_per_us (C++ function), 2062
esp_rom_get_reset_reason (C++ function), 2062
esp_rom_install_channel_putc (C++ function), 2061
esp_rom_install_uart_printf (C++ function), 2061
esp_rom_printf (C++ function), 2061
esp_rom_route_intr_matrix (C++ function), 2062
esp_rrm_is_rrm_supported_connection (C++ function), 938
esp_rrm_send_neighbor_rep_request (C++ function), 938
esp_sco_data_path_t (C++ enum), 531
esp_sco_data_path_t::ESP_SCO_DATA_PATH_HCI (C++ enumerator), 531
esp_sco_data_path_t::ESP_SCO_DATA_PATH_PCM (C++ enumerator), 531
esp_sdp_cb_event_t (C++ enum), 521
esp_sdp_cb_event_t::ESP_SDP_CREATE_RECORD_COMP_EV (C++ enumerator), 521
esp_sdp_cb_event_t::ESP_SDP_DEINIT_EVT (C++ enumerator), 521
esp_sdp_cb_event_t::ESP_SDP_INIT_EVT (C++ enumerator), 521
esp_sdp_cb_event_t::ESP_SDP_REMOVE_RECORD_COMP_EV

(C++ *enumerator*), 522
esp_sdp_cb_event_t::ESP_SDP_SEARCH_COMP_EVT (C++ *enumerator*), 521
esp_sdp_cb_param_t (C++ *union*), 516
esp_sdp_cb_param_t::create_record (C++ *member*), 516
esp_sdp_cb_param_t::deinit (C++ *member*), 516
esp_sdp_cb_param_t::init (C++ *member*), 516
esp_sdp_cb_param_t::remove_record (C++ *member*), 516
esp_sdp_cb_param_t::sdp_crate_record_evt_param (C++ *struct*), 517
esp_sdp_cb_param_t::sdp_crate_record_evt_param::record_hdl (C++ *member*), 517
esp_sdp_cb_param_t::sdp_crate_record_evt_param::status (C++ *member*), 517
esp_sdp_cb_param_t::sdp_deinit_evt_param (C++ *struct*), 517
esp_sdp_cb_param_t::sdp_deinit_evt_param::status (C++ *member*), 517
esp_sdp_cb_param_t::sdp_init_evt_param (C++ *struct*), 517
esp_sdp_cb_param_t::sdp_init_evt_param::status (C++ *member*), 517
esp_sdp_cb_param_t::sdp_remove_record_evt_param (C++ *struct*), 517
esp_sdp_cb_param_t::sdp_remove_record_evt_param::status (C++ *member*), 517
esp_sdp_cb_param_t::sdp_search_evt_param (C++ *struct*), 517
esp_sdp_cb_param_t::sdp_search_evt_param::record_count (C++ *member*), 518
esp_sdp_cb_param_t::sdp_search_evt_param::records (C++ *member*), 518
esp_sdp_cb_param_t::sdp_search_evt_param::remote_addr (C++ *member*), 517
esp_sdp_cb_param_t::sdp_search_evt_param::sdp_record (C++ *member*), 518
esp_sdp_cb_param_t::sdp_search_evt_param::status (C++ *member*), 517
esp_sdp_cb_param_t::search (C++ *member*), 516
esp_sdp_cb_t (C++ *type*), 521
esp_sdp_create_record (C++ *function*), 515
esp_sdp_deinit (C++ *function*), 515
esp_sdp_init (C++ *function*), 515
esp_sdp_register_callback (C++ *function*), 515
esp_sdp_remove_record (C++ *function*), 515
esp_sdp_search_record (C++ *function*), 515
ESP_SDP_SERVER_NAME_MAX (C *macro*), 520
esp_sdp_status_t (C++ *enum*), 521
esp_sdp_status_t::ESP_SDP_FAILURE (C++ *enumerator*), 521
esp_sdp_status_t::ESP_SDP_NEED_DEINIT (C++ *enumerator*), 521
esp_sdp_status_t::ESP_SDP_NEED_INIT (C++ *enumerator*), 521
esp_sdp_status_t::ESP_SDP_NO_CREATE_RECORD (C++ *enumerator*), 521
esp_sdp_status_t::ESP_SDP_NO_RESOURCE (C++ *enumerator*), 521
esp_sdp_status_t::ESP_SDP_SUCCESS (C++ *enumerator*), 521
esp_service_source_t (C++ *enum*), 331
esp_service_source_t::ESP_GATT_SERVICE_FROM_NVS_F (C++ *enumerator*), 331
esp_service_source_t::ESP_GATT_SERVICE_FROM_REMOT (C++ *enumerator*), 331
esp_service_source_t::ESP_GATT_SERVICE_FROM_UNKNO (C++ *enumerator*), 331
esp_set_deep_sleep_wake_stub (C++ *function*), 2131
esp_sleep_config_gpio_isolate (C++ *function*), 2132
esp_sleep_disable_bt_wakeup (C++ *function*), 2130
esp_sleep_disable_wakeup_source (C++ *function*), 2127
esp_sleep_disable_wifi_wakeup (C++ *function*), 2130
esp_sleep_enable_bt_wakeup (C++ *function*), 2130
esp_sleep_enable_ext0_wakeup (C++ *function*), 2128
esp_sleep_enable_ext1_wakeup (C++ *function*), 2129
esp_sleep_enable_gpio_switch (C++ *function*), 2132
esp_sleep_enable_gpio_wakeup (C++ *function*), 2129
esp_sleep_enable_timer_wakeup (C++ *function*), 2127
esp_sleep_enable_touchpad_wakeup (C++ *function*), 2127
esp_sleep_enable_uart_wakeup (C++ *function*), 2129
esp_sleep_enable_ulp_wakeup (C++ *function*), 2127
esp_sleep_enable_wifi_wakeup (C++ *function*), 2130
esp_sleep_ext1_wakeup_mode_t (C++ *enum*), 2132
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ALL (C++ *enumerator*), 2132
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ANY (C++ *enumerator*), 2132
esp_sleep_get_ext1_wakeup_status (C++ *function*), 2130
esp_sleep_get_touchpad_wakeup_status (C++ *function*), 2128
esp_sleep_get_wakeup_cause (C++ *function*), 2131
esp_sleep_is_valid_wakeup_gpio (C++

function), 2128
esp_sleep_pd_config (*C++ function*), 2130
esp_sleep_pd_domain_t (*C++ enum*), 2132
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_MAX (*C++ enumerator*), 2133
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC8M (*C++ enumerator*), 2132
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_FAST_MEM (*C++ enumerator*), 2132
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_PERIPH (*C++ enumerator*), 2132
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_SLOW_MEM (*C++ enumerator*), 2132
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_VDDSDIO (*C++ enumerator*), 2133
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_XTAL (*C++ enumerator*), 2132
esp_sleep_pd_option_t (*C++ enum*), 2133
esp_sleep_pd_option_t::ESP_PD_OPTION_AUTO (*C++ enumerator*), 2133
esp_sleep_pd_option_t::ESP_PD_OPTION_OFF (*C++ enumerator*), 2133
esp_sleep_pd_option_t::ESP_PD_OPTION_ON (*C++ enumerator*), 2133
esp_sleep_source_t (*C++ enum*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_ALL (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_BT (*C++ enumerator*), 2134
esp_sleep_source_t::ESP_SLEEP_WAKEUP_COCPU (*C++ enumerator*), 2134
esp_sleep_source_t::ESP_SLEEP_WAKEUP_COCPU_TRAP_TRIG (*C++ enumerator*), 2134
esp_sleep_source_t::ESP_SLEEP_WAKEUP_EXT0 (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_EXT1 (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_GPIO (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_TIMER (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_TOUCHPAD (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_UART (*C++ enumerator*), 2134
esp_sleep_source_t::ESP_SLEEP_WAKEUP_ULP (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_UNDEFINED (*C++ enumerator*), 2133
esp_sleep_source_t::ESP_SLEEP_WAKEUP_WIFI (*C++ enumerator*), 2134
esp_sleep_wakeup_cause_t (*C++ type*), 2132
esp_smartconfig_fast_mode (*C++ function*), 871
esp_smartconfig_get_rvd_data (*C++ function*), 871
esp_smartconfig_get_version (*C++ function*), 870
esp_smartconfig_set_type (*C++ function*), 871
esp_smartconfig_start (*C++ function*), 870
esp_smartconfig_stop (*C++ function*), 870
esp_sntp_enabled (*C++ function*), 2148
esp_sntp_get_sync_interval (*C macro*), 2148
esp_sntp_get_sync_mode (*C macro*), 2148
esp_sntp_get_sync_status (*C macro*), 2148
esp_sntp_getserver (*C++ function*), 2148
esp_sntp_getservername (*C++ function*), 2148
esp_sntp_init (*C++ function*), 2147
esp_sntp_operatingmode_t (*C++ enum*), 2149
esp_sntp_operatingmode_t::ESP_SNTP_OPMODE_LISTENONLY (*C++ enumerator*), 2149
esp_sntp_operatingmode_t::ESP_SNTP_OPMODE_POLL (*C++ enumerator*), 2149
esp_sntp_restart (*C macro*), 2148
esp_sntp_set_sync_interval (*C macro*), 2148
esp_sntp_set_sync_mode (*C macro*), 2148
esp_sntp_set_sync_status (*C macro*), 2148
esp_sntp_set_time_sync_notification_cb (*C macro*), 2148
esp_sntp_setoperatingmode (*C++ function*), 2147
esp_sntp_setserver (*C++ function*), 2147
esp_sntp_setservername (*C++ function*), 2147
esp_sntp_stop (*C++ function*), 2147
esp_sntp_sync_time (*C macro*), 2148
esp_spiffs_check (*C++ function*), 1806
esp_spiffs_format (*C++ function*), 1806
esp_spiffs_gc (*C++ function*), 1806
esp_spiffs_info (*C++ function*), 1806
esp_spiffs_mounted (*C++ function*), 1806
esp_spp_cb_event_t (*C++ enum*), 442
esp_spp_cb_event_t::ESP_SPP_CL_INIT_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_CLOSE_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_CONG_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_DATA_IND_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_DISCOVERY_COMP_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_INIT_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_OPEN_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_SRV_OPEN_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_SRV_STOP_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_START_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_UNINIT_EVT (*C++ enumerator*), 442
esp_spp_cb_event_t::ESP_SPP_VFS_REGISTER_EVT (*C++ enumerator*), 442

esp_spp_cb_event_t::ESP_SPP_VFS_UNREGISTER_EVT (*C++ enumerator*), 443
esp_spp_cb_event_t::ESP_SPP_WRITE_EVT (*C++ enumerator*), 442
esp_spp_cb_param_t (*C++ union*), 434
esp_spp_cb_param_t::cl_init (*C++ member*), 435
esp_spp_cb_param_t::close (*C++ member*), 435
esp_spp_cb_param_t::cong (*C++ member*), 435
esp_spp_cb_param_t::data_ind (*C++ member*), 435
esp_spp_cb_param_t::disc_comp (*C++ member*), 434
esp_spp_cb_param_t::init (*C++ member*), 434
esp_spp_cb_param_t::open (*C++ member*), 434
esp_spp_cb_param_t::spp_cl_init_evt_param (*C++ struct*), 435
esp_spp_cb_param_t::spp_cl_init_evt_param::fd (*C++ member*), 435
esp_spp_cb_param_t::spp_cl_init_evt_param::handle (*C++ member*), 435
esp_spp_cb_param_t::spp_cl_init_evt_param::sec_id (*C++ member*), 435
esp_spp_cb_param_t::spp_cl_init_evt_param::status (*C++ member*), 435
esp_spp_cb_param_t::spp_close_evt_param (*C++ struct*), 435
esp_spp_cb_param_t::spp_close_evt_param::async (*C++ member*), 436
esp_spp_cb_param_t::spp_close_evt_param::handle (*C++ member*), 436
esp_spp_cb_param_t::spp_close_evt_param::port_status (*C++ member*), 436
esp_spp_cb_param_t::spp_close_evt_param::status (*C++ member*), 436
esp_spp_cb_param_t::spp_cong_evt_param (*C++ struct*), 436
esp_spp_cb_param_t::spp_cong_evt_param::cong (*C++ member*), 436
esp_spp_cb_param_t::spp_cong_evt_param::handle (*C++ member*), 436
esp_spp_cb_param_t::spp_cong_evt_param::status (*C++ member*), 436
esp_spp_cb_param_t::spp_data_ind_evt_param (*C++ struct*), 436
esp_spp_cb_param_t::spp_data_ind_evt_param::data (*C++ member*), 436
esp_spp_cb_param_t::spp_data_ind_evt_param::handle (*C++ member*), 436
esp_spp_cb_param_t::spp_data_ind_evt_param::len (*C++ member*), 436
esp_spp_cb_param_t::spp_data_ind_evt_param::status (*C++ member*), 436
esp_spp_cb_param_t::spp_discovery_comp_evt_param (*C++ struct*), 436
esp_spp_cb_param_t::spp_discovery_comp_evt_param::scn_num (*C++ member*), 437
esp_spp_cb_param_t::spp_discovery_comp_evt_param::scn[17] (*C++ member*), 437
esp_spp_cb_param_t::spp_discovery_comp_evt_param::service_name (*C++ member*), 437
esp_spp_cb_param_t::spp_discovery_comp_evt_param::status (*C++ member*), 437
esp_spp_cb_param_t::spp_init_evt_param (*C++ struct*), 437
esp_spp_cb_param_t::spp_init_evt_param::status (*C++ member*), 437
esp_spp_cb_param_t::spp_open_evt_param (*C++ struct*), 437
esp_spp_cb_param_t::spp_open_evt_param::fd (*C++ member*), 437
esp_spp_cb_param_t::spp_open_evt_param::handle (*C++ member*), 437
esp_spp_cb_param_t::spp_open_evt_param::rem_bda (*C++ member*), 437
esp_spp_cb_param_t::spp_open_evt_param::status (*C++ member*), 437
esp_spp_cb_param_t::spp_srv_open_evt_param (*C++ struct*), 437
esp_spp_cb_param_t::spp_srv_open_evt_param::fd (*C++ member*), 438
esp_spp_cb_param_t::spp_srv_open_evt_param::handle (*C++ member*), 437
esp_spp_cb_param_t::spp_srv_open_evt_param::new_listen_handle (*C++ member*), 438
esp_spp_cb_param_t::spp_srv_open_evt_param::rem_bda (*C++ member*), 438
esp_spp_cb_param_t::spp_srv_open_evt_param::status (*C++ member*), 437
esp_spp_cb_param_t::spp_srv_stop_evt_param (*C++ struct*), 438
esp_spp_cb_param_t::spp_srv_stop_evt_param::scn (*C++ member*), 438
esp_spp_cb_param_t::spp_srv_stop_evt_param::status (*C++ member*), 438
esp_spp_cb_param_t::spp_start_evt_param (*C++ struct*), 438
esp_spp_cb_param_t::spp_start_evt_param::handle (*C++ member*), 438
esp_spp_cb_param_t::spp_start_evt_param::scn (*C++ member*), 438
esp_spp_cb_param_t::spp_start_evt_param::sec_id (*C++ member*), 438
esp_spp_cb_param_t::spp_start_evt_param::status (*C++ member*), 438
esp_spp_cb_param_t::spp_start_evt_param::use_co (*C++ member*), 438
esp_spp_cb_param_t::spp_uninit_evt_param (*C++ struct*), 438
esp_spp_cb_param_t::spp_uninit_evt_param::status (*C++ member*), 439
esp_spp_cb_param_t::spp_vfs_register_evt_param

(C++ *struct*), 439
esp_spp_cb_param_t::spp_vfs_register_evt_param::sec_auth (C++ *member*), 439
esp_spp_cb_param_t::spp_vfs_unregister_evt_param (C++ *struct*), 439
esp_spp_cb_param_t::spp_vfs_unregister_evt_param::status (C++ *member*), 439
esp_spp_cb_param_t::spp_write_evt_param (C++ *struct*), 439
esp_spp_cb_param_t::spp_write_evt_param::cong (C++ *member*), 439
esp_spp_cb_param_t::spp_write_evt_param::handle (C++ *member*), 439
esp_spp_cb_param_t::spp_write_evt_param::len (C++ *member*), 439
esp_spp_cb_param_t::spp_write_evt_param::status (C++ *member*), 439
esp_spp_cb_param_t::srv_open (C++ *member*), 434
esp_spp_cb_param_t::srv_stop (C++ *member*), 435
esp_spp_cb_param_t::start (C++ *member*), 435
esp_spp_cb_param_t::uninit (C++ *member*), 434
esp_spp_cb_param_t::vfs_register (C++ *member*), 435
esp_spp_cb_param_t::vfs_unregister (C++ *member*), 435
esp_spp_cb_param_t::write (C++ *member*), 435
esp_spp_cb_t (C++ *type*), 441
esp_spp_cfg_t (C++ *struct*), 439
esp_spp_cfg_t::enable_l2cap_ertm (C++ *member*), 440
esp_spp_cfg_t::mode (C++ *member*), 440
esp_spp_cfg_t::tx_buffer_size (C++ *member*), 440
esp_spp_connect (C++ *function*), 432
esp_spp_deinit (C++ *function*), 432
esp_spp_disconnect (C++ *function*), 433
esp_spp_enhanced_init (C++ *function*), 432
esp_spp_init (C++ *function*), 432
ESP_SPP_MAX_MTU (C *macro*), 440
ESP_SPP_MAX_SCN (C *macro*), 440
ESP_SPP_MAX_TX_BUFFER_SIZE (C *macro*), 440
ESP_SPP_MIN_TX_BUFFER_SIZE (C *macro*), 440
esp_spp_mode_t (C++ *enum*), 441
esp_spp_mode_t::ESP_SPP_MODE_CB (C++ *enumerator*), 442
esp_spp_mode_t::ESP_SPP_MODE_VFS (C++ *enumerator*), 442
esp_spp_register_callback (C++ *function*), 432
esp_spp_role_t (C++ *enum*), 441
esp_spp_role_t::ESP_SPP_ROLE_MASTER (C++ *enumerator*), 441
esp_spp_role_t::ESP_SPP_ROLE_SLAVE (C++ *enumerator*), 441
ESP_SPP_SEC_AUTHENTICATE (C *macro*), 440
ESP_SPP_SEC_AUTHORIZE (C *macro*), 440
ESP_SPP_SEC_ENCRYPT (C *macro*), 440
ESP_SPP_SEC_IN_16_DIGITS (C *macro*), 440
ESP_SPP_SEC_MITM (C *macro*), 440
ESP_SPP_SEC_MODE4_LEVEL4 (C *macro*), 440
ESP_SPP_SEC_NONE (C *macro*), 440
esp_spp_sec_t (C++ *type*), 441
esp_spp_start_discovery (C++ *function*), 432
esp_spp_start_srv (C++ *function*), 433
esp_spp_status_t (C++ *enum*), 441
esp_spp_status_t::ESP_SPP_BUSY (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_FAILURE (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NEED_DEINIT (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NEED_INIT (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NO_CONNECTION (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NO_DATA (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NO_RESOURCE (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_NO_SERVER (C++ *enumerator*), 441
esp_spp_status_t::ESP_SPP_SUCCESS (C++ *enumerator*), 441
esp_spp_stop_srv (C++ *function*), 433
esp_spp_stop_srv_scn (C++ *function*), 433
esp_spp_vfs_register (C++ *function*), 434
esp_spp_vfs_unregister (C++ *function*), 434
esp_spp_write (C++ *function*), 434
esp_supp_dpp_bootstrap_gen (C++ *function*), 942
esp_supp_dpp_bootstrap_t (C++ *type*), 943
esp_supp_dpp_deinit (C++ *function*), 942
esp_supp_dpp_event_cb_t (C++ *type*), 943
esp_supp_dpp_event_t (C++ *enum*), 943
esp_supp_dpp_event_t::ESP_SUPP_DPP_CFG_RECVD (C++ *enumerator*), 943
esp_supp_dpp_event_t::ESP_SUPP_DPP_FAIL (C++ *enumerator*), 943
esp_supp_dpp_event_t::ESP_SUPP_DPP_URI_READY (C++ *enumerator*), 943
esp_supp_dpp_init (C++ *function*), 941
esp_supp_dpp_start_listen (C++ *function*), 942
esp_supp_dpp_stop_listen (C++ *function*), 942
esp_system_abort (C++ *function*), 2084
esp_sysview_flush (C++ *function*), 1836
esp_sysview_heap_trace_alloc (C++ *function*), 1836
esp_sysview_heap_trace_free (C++ *function*), 1836

esp_sysview_heap_trace_start (C++ function), 1836
esp_sysview_heap_trace_stop (C++ function), 1836
esp_sysview_vprintf (C++ function), 1836
esp_task_wdt_add (C++ function), 2187
esp_task_wdt_add_user (C++ function), 2187
esp_task_wdt_config_t (C++ struct), 2188
esp_task_wdt_config_t::idle_core_mask (C++ member), 2189
esp_task_wdt_config_t::timeout_ms (C++ member), 2189
esp_task_wdt_config_t::trigger_panic (C++ member), 2189
esp_task_wdt_deinit (C++ function), 2187
esp_task_wdt_delete (C++ function), 2188
esp_task_wdt_delete_user (C++ function), 2188
esp_task_wdt_init (C++ function), 2186
esp_task_wdt_isr_user_handler (C++ function), 2188
esp_task_wdt_reconfigure (C++ function), 2186
esp_task_wdt_reset (C++ function), 2187
esp_task_wdt_reset_user (C++ function), 2188
esp_task_wdt_status (C++ function), 2188
esp_task_wdt_user_handle_t (C++ type), 2189
esp_timer_cb_t (C++ type), 2060
esp_timer_create (C++ function), 2057
esp_timer_create_args_t (C++ struct), 2060
esp_timer_create_args_t::arg (C++ member), 2060
esp_timer_create_args_t::callback (C++ member), 2060
esp_timer_create_args_t::dispatch_method (C++ member), 2060
esp_timer_create_args_t::name (C++ member), 2060
esp_timer_create_args_t::skip_unhandled_events (C++ member), 2060
esp_timer_deinit (C++ function), 2057
esp_timer_delete (C++ function), 2058
esp_timer_dispatch_t (C++ enum), 2061
esp_timer_dispatch_t::ESP_TIMER_MAX (C++ enumerator), 2061
esp_timer_dispatch_t::ESP_TIMER_TASK (C++ enumerator), 2061
esp_timer_dump (C++ function), 2059
esp_timer_early_init (C++ function), 2057
esp_timer_get_expiry_time (C++ function), 2059
esp_timer_get_next_alarm (C++ function), 2059
esp_timer_get_next_alarm_for_wake_up (C++ function), 2059
esp_timer_get_period (C++ function), 2059
esp_timer_get_time (C++ function), 2058
esp_timer_handle_t (C++ type), 2060
esp_timer_init (C++ function), 2057
esp_timer_is_active (C++ function), 2060
esp_timer_isr_dispatch_need_yield (C++ function), 2060
esp_timer_restart (C++ function), 2058
esp_timer_start_once (C++ function), 2057
esp_timer_start_periodic (C++ function), 2058
esp_timer_stop (C++ function), 2058
esp_tls_addr_family (C++ enum), 152
esp_tls_addr_family::ESP_TLS_AF_INET (C++ enumerator), 152
esp_tls_addr_family::ESP_TLS_AF_INET6 (C++ enumerator), 152
esp_tls_addr_family::ESP_TLS_AF_UNSPEC (C++ enumerator), 152
esp_tls_addr_family_t (C++ type), 151
esp_tls_cfg (C++ struct), 148
esp_tls_cfg::addr_family (C++ member), 150
esp_tls_cfg::alpn_protos (C++ member), 149
esp_tls_cfg::cacert_buf (C++ member), 149
esp_tls_cfg::cacert_bytes (C++ member), 149
esp_tls_cfg::cacert_pem_buf (C++ member), 149
esp_tls_cfg::cacert_pem_bytes (C++ member), 149
esp_tls_cfg::clientcert_buf (C++ member), 149
esp_tls_cfg::clientcert_bytes (C++ member), 149
esp_tls_cfg::clientcert_pem_buf (C++ member), 149
esp_tls_cfg::clientcert_pem_bytes (C++ member), 149
esp_tls_cfg::clientkey_buf (C++ member), 149
esp_tls_cfg::clientkey_bytes (C++ member), 149
esp_tls_cfg::clientkey_password (C++ member), 149
esp_tls_cfg::clientkey_password_len (C++ member), 150
esp_tls_cfg::clientkey_pem_buf (C++ member), 149
esp_tls_cfg::clientkey_pem_bytes (C++ member), 149
esp_tls_cfg::common_name (C++ member), 150
esp_tls_cfg::crt_bundle_attach (C++ member), 150
esp_tls_cfg::ds_data (C++ member), 150
esp_tls_cfg::if_name (C++ member), 150
esp_tls_cfg::is_plain_tcp (C++ member),

150
esp_tls_cfg::keep_alive_cfg (C++ member), 150
esp_tls_cfg::non_block (C++ member), 150
esp_tls_cfg::psk_hint_key (C++ member), 150
esp_tls_cfg::skip_common_name (C++ member), 150
esp_tls_cfg::timeout_ms (C++ member), 150
esp_tls_cfg::use_global_ca_store (C++ member), 150
esp_tls_cfg::use_secure_element (C++ member), 150
esp_tls_cfg_t (C++ type), 151
esp_tls_conn_destroy (C++ function), 145
esp_tls_conn_http_new (C++ function), 143
esp_tls_conn_http_new_async (C++ function), 145
esp_tls_conn_http_new_sync (C++ function), 144
esp_tls_conn_new_async (C++ function), 144
esp_tls_conn_new_sync (C++ function), 144
esp_tls_conn_read (C++ function), 145
esp_tls_conn_state (C++ enum), 151
esp_tls_conn_state::ESP_TLS_CONNECTING (C++ enumerator), 151
esp_tls_conn_state::ESP_TLS_DONE (C++ enumerator), 151
esp_tls_conn_state::ESP_TLS_FAIL (C++ enumerator), 151
esp_tls_conn_state::ESP_TLS_HANDSHAKE (C++ enumerator), 151
esp_tls_conn_state::ESP_TLS_INIT (C++ enumerator), 151
esp_tls_conn_state_t (C++ type), 151
esp_tls_conn_write (C++ function), 145
ESP_TLS_ERR_SSL_TIMEOUT (C macro), 154
ESP_TLS_ERR_SSL_WANT_READ (C macro), 154
ESP_TLS_ERR_SSL_WANT_WRITE (C macro), 154
esp_tls_error_handle_t (C++ type), 154
esp_tls_error_type_t (C++ enum), 154
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_ESP (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MAX (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS_CERT_FLAGS (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_SYSTEM (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_UNKNOWN (C++ enumerator), 154
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_WOLFSSL (C++ enumerator), 155
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_WOLFSSL_CERT_VERIFY (C++ enumerator), 155
esp_tls_free_global_ca_store (C++ function), 146
esp_tls_get_and_clear_error_type (C++ function), 147
esp_tls_get_and_clear_last_error (C++ function), 146
esp_tls_get_bytes_avail (C++ function), 145
esp_tls_get_conn_sockfd (C++ function), 146
esp_tls_get_error_handle (C++ function), 147
esp_tls_get_global_ca_store (C++ function), 147
esp_tls_get_ssl_context (C++ function), 146
esp_tls_init (C++ function), 143
esp_tls_init_global_ca_store (C++ function), 146
esp_tls_last_error (C++ struct), 152
esp_tls_last_error::esp_tls_error_code (C++ member), 152
esp_tls_last_error::esp_tls_flags (C++ member), 152
esp_tls_last_error::last_error (C++ member), 152
esp_tls_last_error_t (C++ type), 154
esp_tls_plain_tcp_connect (C++ function), 147
esp_tls_role (C++ enum), 151
esp_tls_role::ESP_TLS_CLIENT (C++ enumerator), 151
esp_tls_role::ESP_TLS_SERVER (C++ enumerator), 152
esp_tls_role_t (C++ type), 151
esp_tls_set_global_ca_store (C++ function), 146
esp_tls_t (C++ type), 151
esp_unregister_shutdown_handler (C++ function), 2084
ESP_UUID_LEN_128 (C macro), 238
ESP_UUID_LEN_16 (C macro), 237
ESP_UUID_LEN_32 (C macro), 237
esp_vendor_ie_cb_t (C++ type), 896
esp_vfs_close (C++ function), 1812
esp_vfs_dev_uart_port_set_rx_line_endings (C++ function), 1821
esp_vfs_dev_uart_port_set_tx_line_endings (C++ function), 1821
esp_vfs_dev_uart_register (C++ function), 1820
esp_vfs_dev_uart_set_rx_line_endings (C++ function), 1820
esp_vfs_dev_uart_set_tx_line_endings (C++ function), 1820
esp_vfs_dev_uart_use_driver (C++ function), 1822
esp_vfs_dev_uart_use_nonblocking (C++ function), 1822
ESP_VFS_EVENTD_CONFIG_DEFAULT (C macro), 1823
esp_vfs_eventfd_config_t (C++ struct), 1822

esp_vfs_eventfd_config_t::max_fds (C++ member), 1822
esp_vfs_eventfd_register (C++ function), 1822
esp_vfs_eventfd_unregister (C++ function), 1822
esp_vfs_fat_info (C++ function), 1721
esp_vfs_fat_mount_config_t (C++ struct), 1721
esp_vfs_fat_mount_config_t::allocation_unit_size (C++ member), 1722
esp_vfs_fat_mount_config_t::disk_status_check_enable (C++ member), 1722
esp_vfs_fat_mount_config_t::format_if_mount_failed (C++ member), 1721
esp_vfs_fat_mount_config_t::max_files (C++ member), 1722
esp_vfs_fat_register (C++ function), 1718
esp_vfs_fat_sdcard_unmount (C++ function), 1720
esp_vfs_fat_sdmmc_mount (C++ function), 1718
esp_vfs_fat_sdmmc_mount_config_t (C++ type), 1722
esp_vfs_fat_sdmmc_unmount (C++ function), 1720
esp_vfs_fat_sdspi_mount (C++ function), 1719
esp_vfs_fat_spiflash_mount_ro (C++ function), 1720
esp_vfs_fat_spiflash_mount_rw_wl (C++ function), 1720
esp_vfs_fat_spiflash_unmount_ro (C++ function), 1721
esp_vfs_fat_spiflash_unmount_rw_wl (C++ function), 1720
esp_vfs_fat_unregister_path (C++ function), 1718
ESP_VFS_FLAG_CONTEXT_PTR (C macro), 1820
ESP_VFS_FLAG_DEFAULT (C macro), 1820
esp_vfs_fstat (C++ function), 1812
esp_vfs_id_t (C++ type), 1820
esp_vfs_l2tap_eth_filter (C++ function), 1016
esp_vfs_l2tap_intf_register (C++ function), 1016
esp_vfs_l2tap_intf_unregister (C++ function), 1016
esp_vfs_link (C++ function), 1812
esp_vfs_lseek (C++ function), 1812
esp_vfs_open (C++ function), 1812
ESP_VFS_PATH_MAX (C macro), 1820
esp_vfs_pread (C++ function), 1814
esp_vfs_pwrite (C++ function), 1814
esp_vfs_read (C++ function), 1812
esp_vfs_register (C++ function), 1812
esp_vfs_register_fd (C++ function), 1813
esp_vfs_register_fd_range (C++ function), 1812
esp_vfs_register_fd_with_local_fd (C++ function), 1813
esp_vfs_register_with_id (C++ function), 1812
esp_vfs_rename (C++ function), 1812
esp_vfs_select (C++ function), 1813
esp_vfs_select_sem_t (C++ struct), 1814
esp_vfs_select_sem_t::is_sem_local (C++ member), 1814
esp_vfs_select_sem_t::sem (C++ member), 1814
esp_vfs_select_triggered (C++ function), 1814
esp_vfs_select_triggered_isr (C++ function), 1814
esp_vfs_spiffs_conf_t (C++ struct), 1807
esp_vfs_spiffs_conf_t::base_path (C++ member), 1807
esp_vfs_spiffs_conf_t::format_if_mount_failed (C++ member), 1807
esp_vfs_spiffs_conf_t::max_files (C++ member), 1807
esp_vfs_spiffs_conf_t::partition_label (C++ member), 1807
esp_vfs_spiffs_register (C++ function), 1806
esp_vfs_spiffs_unregister (C++ function), 1806
esp_vfs_stat (C++ function), 1812
esp_vfs_t (C++ struct), 1815
esp_vfs_t::access (C++ member), 1818
esp_vfs_t::access_p (C++ member), 1818
esp_vfs_t::close (C++ member), 1816
esp_vfs_t::close_p (C++ member), 1816
esp_vfs_t::closedir (C++ member), 1817
esp_vfs_t::closedir_p (C++ member), 1817
esp_vfs_t::end_select (C++ member), 1820
esp_vfs_t::fcntl (C++ member), 1817
esp_vfs_t::fcntl_p (C++ member), 1817
esp_vfs_t::flags (C++ member), 1815
esp_vfs_t::fstat (C++ member), 1816
esp_vfs_t::fstat_p (C++ member), 1816
esp_vfs_t::fsync (C++ member), 1818
esp_vfs_t::fsync_p (C++ member), 1818
esp_vfs_t::ftruncate (C++ member), 1818
esp_vfs_t::ftruncate_p (C++ member), 1818
esp_vfs_t::get_socket_select_semaphore (C++ member), 1819
esp_vfs_t::ioctl (C++ member), 1818
esp_vfs_t::ioctl_p (C++ member), 1818
esp_vfs_t::link (C++ member), 1816
esp_vfs_t::link_p (C++ member), 1816
esp_vfs_t::lseek (C++ member), 1815
esp_vfs_t::lseek_p (C++ member), 1815
esp_vfs_t::mkdir (C++ member), 1817
esp_vfs_t::mkdir_p (C++ member), 1817
esp_vfs_t::open (C++ member), 1816

esp_vfs_t::open_p (*C++ member*), 1816
esp_vfs_t::opendir (*C++ member*), 1816
esp_vfs_t::opendir_p (*C++ member*), 1816
esp_vfs_t::pread (*C++ member*), 1815
esp_vfs_t::pread_p (*C++ member*), 1815
esp_vfs_t::pwrite (*C++ member*), 1815
esp_vfs_t::pwrite_p (*C++ member*), 1815
esp_vfs_t::read (*C++ member*), 1815
esp_vfs_t::read_p (*C++ member*), 1815
esp_vfs_t::readdir (*C++ member*), 1817
esp_vfs_t::readdir_p (*C++ member*), 1817
esp_vfs_t::readdir_r (*C++ member*), 1817
esp_vfs_t::readdir_r_p (*C++ member*), 1817
esp_vfs_t::rename (*C++ member*), 1816
esp_vfs_t::rename_p (*C++ member*), 1816
esp_vfs_t::rmdir (*C++ member*), 1817
esp_vfs_t::rmdir_p (*C++ member*), 1817
esp_vfs_t::seekdir (*C++ member*), 1817
esp_vfs_t::seekdir_p (*C++ member*), 1817
esp_vfs_t::socket_select (*C++ member*), 1819
esp_vfs_t::start_select (*C++ member*), 1819
esp_vfs_t::stat (*C++ member*), 1816
esp_vfs_t::stat_p (*C++ member*), 1816
esp_vfs_t::stop_socket_select (*C++ member*), 1819
esp_vfs_t::stop_socket_select_isr (*C++ member*), 1819
esp_vfs_t::tcdrain (*C++ member*), 1819
esp_vfs_t::tcdrain_p (*C++ member*), 1819
esp_vfs_t::tcflow (*C++ member*), 1819
esp_vfs_t::tcflow_p (*C++ member*), 1819
esp_vfs_t::tcflush (*C++ member*), 1819
esp_vfs_t::tcflush_p (*C++ member*), 1819
esp_vfs_t::tcgetattr (*C++ member*), 1818
esp_vfs_t::tcgetattr_p (*C++ member*), 1818
esp_vfs_t::tcgetsid (*C++ member*), 1819
esp_vfs_t::tcgetsid_p (*C++ member*), 1819
esp_vfs_t::tcsendbreak (*C++ member*), 1819
esp_vfs_t::tcsendbreak_p (*C++ member*), 1819
esp_vfs_t::tcsetattr (*C++ member*), 1818
esp_vfs_t::tcsetattr_p (*C++ member*), 1818
esp_vfs_t::telldir (*C++ member*), 1817
esp_vfs_t::telldir_p (*C++ member*), 1817
esp_vfs_t::truncate (*C++ member*), 1818
esp_vfs_t::truncate_p (*C++ member*), 1818
esp_vfs_t::unlink (*C++ member*), 1816
esp_vfs_t::unlink_p (*C++ member*), 1816
esp_vfs_t::utime (*C++ member*), 1818
esp_vfs_t::utime_p (*C++ member*), 1818
esp_vfs_t::write (*C++ member*), 1815
esp_vfs_t::write_p (*C++ member*), 1815
esp_vfs_unlink (*C++ function*), 1812
esp_vfs_unregister (*C++ function*), 1813
esp_vfs_unregister_fd (*C++ function*), 1813
esp_vfs_unregister_with_id (*C++ function*), 1813
esp_vfs_usb_serial_jtag_use_driver (*C++ function*), 1822
esp_vfs_usb_serial_jtag_use_nonblocking (*C++ function*), 1822
esp_vfs_utime (*C++ function*), 1812
esp_vfs_write (*C++ function*), 1812
esp_vhci_host_callback (*C++ struct*), 528
esp_vhci_host_callback::notify_host_recv (*C++ member*), 528
esp_vhci_host_callback::notify_host_send_availabl (*C++ member*), 528
esp_vhci_host_callback_t (*C++ type*), 528
esp_vhci_host_check_send_available (*C++ function*), 524
esp_vhci_host_register_callback (*C++ function*), 524
esp_vhci_host_send_packet (*C++ function*), 524
esp_wake_deep_sleep (*C++ function*), 2131
esp_wifi_80211_tx (*C++ function*), 885
esp_wifi_ap_get_sta_aid (*C++ function*), 883
esp_wifi_ap_get_sta_list (*C++ function*), 883
esp_wifi_ap_wps_disable (*C++ function*), 936
esp_wifi_ap_wps_enable (*C++ function*), 935
esp_wifi_ap_wps_start (*C++ function*), 936
esp_wifi_bt_power_domain_off (*C++ function*), 526
esp_wifi_bt_power_domain_on (*C++ function*), 526
esp_wifi_clear_ap_list (*C++ function*), 877
esp_wifi_clear_default_wifi_driver_and_handlers (*C++ function*), 1017
esp_wifi_clear_fast_connect (*C++ function*), 876
esp_wifi_config_11b_rate (*C++ function*), 889
esp_wifi_config_80211_tx_rate (*C++ function*), 890
esp_wifi_config_espnow_rate (*C++ function*), 827
esp_wifi_connect (*C++ function*), 875
ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE (*C macro*), 895
esp_wifi_connectionless_module_set_wake_interval (*C++ function*), 889
esp_wifi_deauth_sta (*C++ function*), 876
esp_wifi_deinit (*C++ function*), 874
esp_wifi_disable_pmf_config (*C++ function*), 891
esp_wifi_disconnect (*C++ function*), 875
esp_wifi_force_wakeup_acquire (*C++ function*), 889
esp_wifi_force_wakeup_release (*C++ function*), 889
esp_wifi_ftm_end_session (*C++ function*),

888
esp_wifi_ftm_get_report (C++ function), 888
esp_wifi_ftm_initiate_session (C++ function), 888
esp_wifi_ftm_resp_set_offset (C++ function), 888
esp_wifi_get_ant (C++ function), 887
esp_wifi_get_ant_gpio (C++ function), 886
esp_wifi_get_bandwidth (C++ function), 879
esp_wifi_get_channel (C++ function), 880
esp_wifi_get_config (C++ function), 883
esp_wifi_get_country (C++ function), 880
esp_wifi_get_country_code (C++ function), 890
esp_wifi_get_event_mask (C++ function), 885
esp_wifi_get_inactive_time (C++ function), 887
esp_wifi_get_mac (C++ function), 881
esp_wifi_get_max_tx_power (C++ function), 884
esp_wifi_get_mode (C++ function), 874
esp_wifi_get_promiscuous (C++ function), 881
esp_wifi_get_promiscuous_ctrl_filter (C++ function), 882
esp_wifi_get_promiscuous_filter (C++ function), 882
esp_wifi_get_protocol (C++ function), 878
esp_wifi_get_ps (C++ function), 878
esp_wifi_get_tsf_time (C++ function), 887
esp_wifi_init (C++ function), 874
ESP_WIFI_MAX_CONN_NUM (C macro), 914
esp_wifi_restore (C++ function), 875
esp_wifi_scan_get_ap_num (C++ function), 876
esp_wifi_scan_get_ap_record (C++ function), 877
esp_wifi_scan_get_ap_records (C++ function), 877
esp_wifi_scan_start (C++ function), 876
esp_wifi_scan_stop (C++ function), 876
esp_wifi_set_ant (C++ function), 886
esp_wifi_set_ant_gpio (C++ function), 886
esp_wifi_set_bandwidth (C++ function), 879
esp_wifi_set_channel (C++ function), 879
esp_wifi_set_config (C++ function), 882
esp_wifi_set_country (C++ function), 880
esp_wifi_set_country_code (C++ function), 890
esp_wifi_set_csi (C++ function), 886
esp_wifi_set_csi_config (C++ function), 886
esp_wifi_set_csi_rx_cb (C++ function), 885
esp_wifi_set_default_wifi_ap_handlers (C++ function), 1017
esp_wifi_set_default_wifi_sta_handlers (C++ function), 1017
esp_wifi_set_event_mask (C++ function), 885
esp_wifi_set_inactive_time (C++ function), 887
esp_wifi_set_mac (C++ function), 881
esp_wifi_set_max_tx_power (C++ function), 884
esp_wifi_set_mode (C++ function), 874
esp_wifi_set_promiscuous (C++ function), 881
esp_wifi_set_promiscuous_ctrl_filter (C++ function), 882
esp_wifi_set_promiscuous_filter (C++ function), 881
esp_wifi_set_promiscuous_rx_cb (C++ function), 881
esp_wifi_set_protocol (C++ function), 878
esp_wifi_set_ps (C++ function), 878
esp_wifi_set_rssi_threshold (C++ function), 888
esp_wifi_set_storage (C++ function), 883
esp_wifi_set_vendor_ie (C++ function), 884
esp_wifi_set_vendor_ie_cb (C++ function), 884
esp_wifi_sta_enterprise_disable (C++ function), 930
esp_wifi_sta_enterprise_enable (C++ function), 930
esp_wifi_sta_get_aid (C++ function), 891
esp_wifi_sta_get_ap_info (C++ function), 878
esp_wifi_sta_get_negotiated_phymode (C++ function), 891
esp_wifi_sta_get_rssi (C++ function), 891
esp_wifi_start (C++ function), 875
esp_wifi_statis_dump (C++ function), 887
esp_wifi_stop (C++ function), 875
esp_wifi_wps_disable (C++ function), 935
esp_wifi_wps_enable (C++ function), 935
esp_wifi_wps_start (C++ function), 935
esp_wnm_is_btm_supported_connection (C++ function), 939
esp_wnm_send_bss_transition_mgmt_query (C++ function), 939
esp_wps_config_t (C++ struct), 936
esp_wps_config_t::factory_info (C++ member), 937
esp_wps_config_t::pin (C++ member), 937
esp_wps_config_t::wps_type (C++ member), 937
essl_clear_intr (C++ function), 188
essl_get_intr (C++ function), 188
essl_get_intr_ena (C++ function), 189
essl_get_packet (C++ function), 187
essl_get_rx_data_size (C++ function), 187
essl_get_tx_buffer_num (C++ function), 186
essl_handle_t (C++ type), 189
essl_init (C++ function), 186
essl_read_reg (C++ function), 188
essl_reset_cnt (C++ function), 187
essl_sdio_config_t (C++ struct), 190

essl_sdio_config_t::card (C++ member), 190
essl_sdio_config_t::recv_buffer_size (C++ member), 190
essl_sdio_deinit_dev (C++ function), 190
essl_sdio_init_dev (C++ function), 189
essl_send_packet (C++ function), 187
essl_send_slave_intr (C++ function), 189
essl_set_intr_ena (C++ function), 189
essl_spi_config_t (C++ struct), 195
essl_spi_config_t::rx_sync_reg (C++ member), 196
essl_spi_config_t::spi (C++ member), 195
essl_spi_config_t::tx_buf_size (C++ member), 195
essl_spi_config_t::tx_sync_reg (C++ member), 195
essl_spi_deinit_dev (C++ function), 190
essl_spi_get_packet (C++ function), 191
essl_spi_init_dev (C++ function), 190
essl_spi_rdbuf (C++ function), 192
essl_spi_rdbuf_polling (C++ function), 192
essl_spi_rddma (C++ function), 193
essl_spi_rddma_done (C++ function), 194
essl_spi_rddma_seg (C++ function), 194
essl_spi_read_reg (C++ function), 190
essl_spi_reset_cnt (C++ function), 192
essl_spi_send_packet (C++ function), 192
essl_spi_wrbuf (C++ function), 193
essl_spi_wrbuf_polling (C++ function), 193
essl_spi_wrdma (C++ function), 194
essl_spi_wrdma_done (C++ function), 195
essl_spi_wrdma_seg (C++ function), 195
essl_spi_write_reg (C++ function), 191
essl_wait_for_ready (C++ function), 186
essl_wait_int (C++ function), 188
essl_write_reg (C++ function), 187
eTaskGetState (C++ function), 1911
eTaskState (C++ enum), 1934
eTaskState::eBlocked (C++ enumerator), 1934
eTaskState::eDeleted (C++ enumerator), 1934
eTaskState::eInvalid (C++ enumerator), 1934
eTaskState::eReady (C++ enumerator), 1934
eTaskState::eRunning (C++ enumerator), 1934
eTaskState::eSuspended (C++ enumerator), 1934
ETH_DEFAULT_CONFIG (C macro), 959
eth_esp32_emac_config_t (C++ struct), 967
eth_esp32_emac_config_t::clock_config (C++ member), 967
eth_esp32_emac_config_t::dma_burst_len (C++ member), 967
eth_esp32_emac_config_t::interface (C++ member), 967
eth_esp32_emac_config_t::smi_mdc_gpio_num (C++ member), 967
eth_esp32_emac_config_t::smi_mdio_gpio_num (C++ member), 967
ETH_ESP32_EMAC_DEFAULT_CONFIG (C macro), 968
eth_event_t (C++ enum), 961
eth_event_t::ETHERNET_EVENT_CONNECTED (C++ enumerator), 962
eth_event_t::ETHERNET_EVENT_DISCONNECTED (C++ enumerator), 962
eth_event_t::ETHERNET_EVENT_START (C++ enumerator), 962
eth_event_t::ETHERNET_EVENT_STOP (C++ enumerator), 962
eth_mac_clock_config_t (C++ union), 962
eth_mac_clock_config_t::clock_gpio (C++ member), 962
eth_mac_clock_config_t::clock_mode (C++ member), 962
eth_mac_clock_config_t::mii (C++ member), 962
eth_mac_clock_config_t::rmii (C++ member), 962
eth_mac_config_t (C++ struct), 967
eth_mac_config_t::flags (C++ member), 967
eth_mac_config_t::rx_task_prio (C++ member), 967
eth_mac_config_t::rx_task_stack_size (C++ member), 967
eth_mac_config_t::sw_reset_timeout_ms (C++ member), 967
ETH_MAC_DEFAULT_CONFIG (C macro), 968
ETH_MAC_FLAG_PIN_TO_CORE (C macro), 968
ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE (C macro), 968
eth_phy_autoneg_cmd_t (C++ enum), 974
eth_phy_autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_DIS (C++ enumerator), 974
eth_phy_autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_EN (C++ enumerator), 974
eth_phy_autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_G_STA (C++ enumerator), 974
eth_phy_autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_RESTA (C++ enumerator), 974
eth_phy_config_t (C++ struct), 973
eth_phy_config_t::autonego_timeout_ms (C++ member), 973
eth_phy_config_t::phy_addr (C++ member), 973
eth_phy_config_t::reset_gpio_num (C++ member), 973
eth_phy_config_t::reset_timeout_ms (C++ member), 973
ETH_PHY_DEFAULT_CONFIG (C macro), 973
ETS_INTERNAL_INTR_SOURCE_OFF (C macro), 2073
ETS_INTERNAL_PROFILING_INTR_SOURCE (C macro), 2073
ETS_INTERNAL_SW0_INTR_SOURCE (C macro), 2073
ETS_INTERNAL_SW1_INTR_SOURCE (C macro),

2073
ETS_INTERNAL_TIMER0_INTR_SOURCE (C macro), 2073
ETS_INTERNAL_TIMER1_INTR_SOURCE (C macro), 2073
ETS_INTERNAL_TIMER2_INTR_SOURCE (C macro), 2073
ETS_INTERNAL_UNUSED_INTR_SOURCE (C macro), 2073
EventBits_t (C++ type), 1994
eventfd (C++ function), 1822
EventGroupHandle_t (C++ type), 1994
EXT_ADV_TX_PWR_NO_PREFERENCE (C macro), 302

F

ff_diskio_impl_t (C++ struct), 1716
ff_diskio_impl_t::init (C++ member), 1716
ff_diskio_impl_t::ioctl (C++ member), 1716
ff_diskio_impl_t::read (C++ member), 1716
ff_diskio_impl_t::status (C++ member), 1716
ff_diskio_impl_t::write (C++ member), 1716
ff_diskio_register (C++ function), 1715
ff_diskio_register_raw_partition (C++ function), 1716
ff_diskio_register_sdmmc (C++ function), 1716
ff_diskio_register_wl_partition (C++ function), 1716
filter_cb_t (C++ type), 1336

G

get_phy_version_str (C++ function), 2439
gpio_config (C++ function), 1054
gpio_config_t (C++ struct), 1060
gpio_config_t::intr_type (C++ member), 1060
gpio_config_t::mode (C++ member), 1060
gpio_config_t::pin_bit_mask (C++ member), 1060
gpio_config_t::pull_down_en (C++ member), 1060
gpio_config_t::pull_up_en (C++ member), 1060
gpio_deep_sleep_hold_dis (C++ function), 1059
gpio_deep_sleep_hold_en (C++ function), 1059
gpio_drive_cap_t (C++ enum), 1067
gpio_drive_cap_t::GPIO_DRIVE_CAP_0 (C++ enumerator), 1067
gpio_drive_cap_t::GPIO_DRIVE_CAP_1 (C++ enumerator), 1067
gpio_drive_cap_t::GPIO_DRIVE_CAP_2 (C++ enumerator), 1067
gpio_drive_cap_t::GPIO_DRIVE_CAP_3 (C++ enumerator), 1068
gpio_drive_cap_t::GPIO_DRIVE_CAP_DEFAULT (C++ enumerator), 1067
gpio_drive_cap_t::GPIO_DRIVE_CAP_MAX (C++ enumerator), 1068
gpio_get_drive_capability (C++ function), 1058
gpio_get_level (C++ function), 1055
gpio_hold_dis (C++ function), 1058
gpio_hold_en (C++ function), 1058
gpio_install_isr_service (C++ function), 1057
gpio_int_type_t (C++ enum), 1066
gpio_int_type_t::GPIO_INTR_ANYEDGE (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_DISABLE (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_HIGH_LEVEL (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_LOW_LEVEL (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_MAX (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_NEGEDGE (C++ enumerator), 1066
gpio_int_type_t::GPIO_INTR_POSEDGE (C++ enumerator), 1066
gpio_intr_disable (C++ function), 1055
gpio_intr_enable (C++ function), 1054
gpio_iomux_in (C++ function), 1059
gpio_iomux_out (C++ function), 1059
GPIO_IS_VALID_DIGITAL_IO_PAD (C macro), 1060
GPIO_IS_VALID_GPIO (C macro), 1060
GPIO_IS_VALID_OUTPUT_GPIO (C macro), 1060
gpio_isr_handle_t (C++ type), 1061
gpio_isr_handler_add (C++ function), 1057
gpio_isr_handler_remove (C++ function), 1058
gpio_isr_register (C++ function), 1056
gpio_isr_t (C++ type), 1061
gpio_mode_t (C++ enum), 1066
gpio_mode_t::GPIO_MODE_DISABLE (C++ enumerator), 1066
gpio_mode_t::GPIO_MODE_INPUT (C++ enumerator), 1066
gpio_mode_t::GPIO_MODE_INPUT_OUTPUT (C++ enumerator), 1066
gpio_mode_t::GPIO_MODE_INPUT_OUTPUT_OD (C++ enumerator), 1066
gpio_mode_t::GPIO_MODE_OUTPUT (C++ enumerator), 1066
gpio_mode_t::GPIO_MODE_OUTPUT_OD (C++ enumerator), 1066
gpio_num_t (C++ enum), 1063
gpio_num_t::GPIO_NUM_0 (C++ enumerator), 1063

gpio_num_t::GPIO_NUM_1 (C++ enumerator), 1063
gpio_num_t::GPIO_NUM_10 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_11 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_12 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_13 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_14 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_15 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_16 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_17 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_18 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_19 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_2 (C++ enumerator), 1063
gpio_num_t::GPIO_NUM_20 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_21 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_22 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_23 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_25 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_26 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_27 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_28 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_29 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_3 (C++ enumerator), 1063
gpio_num_t::GPIO_NUM_30 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_31 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_32 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_33 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_34 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_35 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_36 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_37 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_38 (C++ enumerator), 1065
gpio_num_t::GPIO_NUM_39 (C++ enumerator), 1066
gpio_num_t::GPIO_NUM_4 (C++ enumerator), 1063
gpio_num_t::GPIO_NUM_5 (C++ enumerator), 1063
gpio_num_t::GPIO_NUM_6 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_7 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_8 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_9 (C++ enumerator), 1064
gpio_num_t::GPIO_NUM_MAX (C++ enumerator), 1066
gpio_num_t::GPIO_NUM_NC (C++ enumerator), 1063
GPIO_PIN_COUNT (C macro), 1060
GPIO_PIN_REG_0 (C macro), 1061
GPIO_PIN_REG_1 (C macro), 1061
GPIO_PIN_REG_10 (C macro), 1061
GPIO_PIN_REG_11 (C macro), 1061
GPIO_PIN_REG_12 (C macro), 1061
GPIO_PIN_REG_13 (C macro), 1061
GPIO_PIN_REG_14 (C macro), 1061
GPIO_PIN_REG_15 (C macro), 1061
GPIO_PIN_REG_16 (C macro), 1061
GPIO_PIN_REG_17 (C macro), 1061
GPIO_PIN_REG_18 (C macro), 1062
GPIO_PIN_REG_19 (C macro), 1062
GPIO_PIN_REG_2 (C macro), 1061
GPIO_PIN_REG_20 (C macro), 1062
GPIO_PIN_REG_21 (C macro), 1062
GPIO_PIN_REG_22 (C macro), 1062
GPIO_PIN_REG_23 (C macro), 1062
GPIO_PIN_REG_24 (C macro), 1062
GPIO_PIN_REG_25 (C macro), 1062
GPIO_PIN_REG_26 (C macro), 1062
GPIO_PIN_REG_27 (C macro), 1062
GPIO_PIN_REG_28 (C macro), 1062
GPIO_PIN_REG_29 (C macro), 1062
GPIO_PIN_REG_3 (C macro), 1061
GPIO_PIN_REG_30 (C macro), 1062
GPIO_PIN_REG_31 (C macro), 1062
GPIO_PIN_REG_32 (C macro), 1062
GPIO_PIN_REG_33 (C macro), 1062
GPIO_PIN_REG_34 (C macro), 1062
GPIO_PIN_REG_35 (C macro), 1062
GPIO_PIN_REG_36 (C macro), 1062
GPIO_PIN_REG_37 (C macro), 1062
GPIO_PIN_REG_38 (C macro), 1062
GPIO_PIN_REG_39 (C macro), 1062
GPIO_PIN_REG_4 (C macro), 1061

GPIO_PIN_REG_40 (*C macro*), 1062
GPIO_PIN_REG_41 (*C macro*), 1063
GPIO_PIN_REG_42 (*C macro*), 1063
GPIO_PIN_REG_43 (*C macro*), 1063
GPIO_PIN_REG_44 (*C macro*), 1063
GPIO_PIN_REG_45 (*C macro*), 1063
GPIO_PIN_REG_46 (*C macro*), 1063
GPIO_PIN_REG_47 (*C macro*), 1063
GPIO_PIN_REG_48 (*C macro*), 1063
GPIO_PIN_REG_5 (*C macro*), 1061
GPIO_PIN_REG_6 (*C macro*), 1061
GPIO_PIN_REG_7 (*C macro*), 1061
GPIO_PIN_REG_8 (*C macro*), 1061
GPIO_PIN_REG_9 (*C macro*), 1061
gpio_port_t (*C++ enum*), 1063
gpio_port_t::GPIO_PORT_0 (*C++ enumerator*), 1063
gpio_port_t::GPIO_PORT_MAX (*C++ enumerator*), 1063
gpio_pull_mode_t (*C++ enum*), 1067
gpio_pull_mode_t::GPIO_FLOATING (*C++ enumerator*), 1067
gpio_pull_mode_t::GPIO_PULLDOWN_ONLY (*C++ enumerator*), 1067
gpio_pull_mode_t::GPIO_PULLUP_ONLY (*C++ enumerator*), 1067
gpio_pull_mode_t::GPIO_PULLUP_PULLDOWN (*C++ enumerator*), 1067
gpio_pulldown_dis (*C++ function*), 1057
gpio_pulldown_en (*C++ function*), 1057
gpio_pulldown_t (*C++ enum*), 1067
gpio_pulldown_t::GPIO_PULLDOWN_DISABLE (*C++ enumerator*), 1067
gpio_pulldown_t::GPIO_PULLDOWN_ENABLE (*C++ enumerator*), 1067
gpio_pullup_dis (*C++ function*), 1057
gpio_pullup_en (*C++ function*), 1056
gpio_pullup_t (*C++ enum*), 1067
gpio_pullup_t::GPIO_PULLUP_DISABLE (*C++ enumerator*), 1067
gpio_pullup_t::GPIO_PULLUP_ENABLE (*C++ enumerator*), 1067
gpio_reset_pin (*C++ function*), 1054
gpio_set_direction (*C++ function*), 1055
gpio_set_drive_capability (*C++ function*), 1058
gpio_set_intr_type (*C++ function*), 1054
gpio_set_level (*C++ function*), 1055
gpio_set_pull_mode (*C++ function*), 1056
gpio_sleep_sel_dis (*C++ function*), 1059
gpio_sleep_sel_en (*C++ function*), 1059
gpio_sleep_set_direction (*C++ function*), 1059
gpio_sleep_set_pull_mode (*C++ function*), 1060
gpio_uninstall_isr_service (*C++ function*), 1057
gpio_wakeup_disable (*C++ function*), 1056
gpio_wakeup_enable (*C++ function*), 1056
gptimer_alarm_cb_t (*C++ type*), 1083
gptimer_alarm_config_t (*C++ struct*), 1082
gptimer_alarm_config_t::alarm_count (*C++ member*), 1082
gptimer_alarm_config_t::auto_reload_on_alarm (*C++ member*), 1083
gptimer_alarm_config_t::flags (*C++ member*), 1083
gptimer_alarm_config_t::reload_count (*C++ member*), 1082
gptimer_alarm_event_data_t (*C++ struct*), 1081
gptimer_alarm_event_data_t::alarm_value (*C++ member*), 1081
gptimer_alarm_event_data_t::count_value (*C++ member*), 1081
gptimer_clock_source_t (*C++ type*), 1083
gptimer_config_t (*C++ struct*), 1082
gptimer_config_t::clk_src (*C++ member*), 1082
gptimer_config_t::direction (*C++ member*), 1082
gptimer_config_t::flags (*C++ member*), 1082
gptimer_config_t::intr_priority (*C++ member*), 1082
gptimer_config_t::intr_shared (*C++ member*), 1082
gptimer_config_t::resolution_hz (*C++ member*), 1082
gptimer_count_direction_t (*C++ enum*), 1083
gptimer_count_direction_t::GPTIMER_COUNT_DOWN (*C++ enumerator*), 1083
gptimer_count_direction_t::GPTIMER_COUNT_UP (*C++ enumerator*), 1083
gptimer_del_timer (*C++ function*), 1078
gptimer_disable (*C++ function*), 1080
gptimer_enable (*C++ function*), 1079
gptimer_event_callbacks_t (*C++ struct*), 1081
gptimer_event_callbacks_t::on_alarm (*C++ member*), 1082
gptimer_get_raw_count (*C++ function*), 1078
gptimer_handle_t (*C++ type*), 1083
gptimer_new_timer (*C++ function*), 1077
gptimer_register_event_callbacks (*C++ function*), 1079
gptimer_set_alarm_action (*C++ function*), 1079
gptimer_set_raw_count (*C++ function*), 1078
gptimer_start (*C++ function*), 1080
gptimer_stop (*C++ function*), 1081

H

heap_caps_add_region (*C++ function*), 2039

heap_caps_add_region_with_caps (*C++ function*), 2040
heap_caps_aligned_alloc (*C++ function*), 2034
heap_caps_aligned_calloc (*C++ function*), 2034
heap_caps_aligned_free (*C++ function*), 2034
heap_caps_calloc (*C++ function*), 2034
heap_caps_calloc_prefer (*C++ function*), 2037
heap_caps_check_integrity (*C++ function*), 2036
heap_caps_check_integrity_addr (*C++ function*), 2036
heap_caps_check_integrity_all (*C++ function*), 2036
heap_caps_dump (*C++ function*), 2037
heap_caps_dump_all (*C++ function*), 2037
heap_caps_enable_nonos_stack_heaps (*C++ function*), 2039
heap_caps_free (*C++ function*), 2033
heap_caps_get_allocated_size (*C++ function*), 2037
heap_caps_get_free_size (*C++ function*), 2035
heap_caps_get_info (*C++ function*), 2035
heap_caps_get_largest_free_block (*C++ function*), 2035
heap_caps_get_minimum_free_size (*C++ function*), 2035
heap_caps_get_total_size (*C++ function*), 2035
heap_caps_init (*C++ function*), 2039
heap_caps_malloc (*C++ function*), 2033
heap_caps_malloc_extmem_enable (*C++ function*), 2036
heap_caps_malloc_prefer (*C++ function*), 2036
heap_caps_print_heap_info (*C++ function*), 2035
heap_caps_realloc (*C++ function*), 2033
heap_caps_realloc_prefer (*C++ function*), 2037
heap_caps_register_failed_alloc_callback (*C++ function*), 2033
heap_trace_dump (*C++ function*), 2053
heap_trace_get (*C++ function*), 2053
heap_trace_get_count (*C++ function*), 2053
heap_trace_init_standalone (*C++ function*), 2052
heap_trace_init_tohost (*C++ function*), 2052
heap_trace_mode_t (*C++ enum*), 2054
heap_trace_mode_t::HEAP_TRACE_ALL (*C++ enumerator*), 2054
heap_trace_mode_t::HEAP_TRACE_LEAKS (*C++ enumerator*), 2054
heap_trace_record_t (*C++ struct*), 2054
heap_trace_record_t::address (*C++ member*), 2054
heap_trace_record_t::alloced_by (*C++ member*), 2054
heap_trace_record_t::ccount (*C++ member*), 2054
heap_trace_record_t::freed_by (*C++ member*), 2054
heap_trace_record_t::size (*C++ member*), 2054
heap_trace_resume (*C++ function*), 2053
heap_trace_start (*C++ function*), 2052
heap_trace_stop (*C++ function*), 2053
http_client_init_cb_t (*C++ type*), 1878
http_event_handle_cb (*C++ type*), 167
HTTPD_200 (*C macro*), 220
HTTPD_204 (*C macro*), 220
HTTPD_207 (*C macro*), 220
HTTPD_400 (*C macro*), 220
HTTPD_404 (*C macro*), 220
HTTPD_408 (*C macro*), 220
HTTPD_500 (*C macro*), 220
httpd_close_func_t (*C++ type*), 223
httpd_config (*C++ struct*), 216
httpd_config::backlog_conn (*C++ member*), 217
httpd_config::close_fn (*C++ member*), 218
httpd_config::core_id (*C++ member*), 216
httpd_config::ctrl_port (*C++ member*), 216
httpd_config::enable_so_linger (*C++ member*), 217
httpd_config::global_transport_ctx (*C++ member*), 217
httpd_config::global_transport_ctx_free_fn (*C++ member*), 217
httpd_config::global_user_ctx (*C++ member*), 217
httpd_config::global_user_ctx_free_fn (*C++ member*), 217
httpd_config::keep_alive_count (*C++ member*), 218
httpd_config::keep_alive_enable (*C++ member*), 217
httpd_config::keep_alive_idle (*C++ member*), 218
httpd_config::keep_alive_interval (*C++ member*), 218
httpd_config::linger_timeout (*C++ member*), 217
httpd_config::lru_purge_enable (*C++ member*), 217
httpd_config::max_open_sockets (*C++ member*), 217
httpd_config::max_resp_headers (*C++ member*), 217
httpd_config::max_uri_handlers (*C++ member*), 217
httpd_config::open_fn (*C++ member*), 218
httpd_config::recv_wait_timeout (*C++*

member), 217
httpd_config::send_wait_timeout (*C++ member*), 217
httpd_config::server_port (*C++ member*), 216
httpd_config::stack_size (*C++ member*), 216
httpd_config::task_priority (*C++ member*), 216
httpd_config::uri_match_fn (*C++ member*), 218
httpd_config_t (*C++ type*), 224
HTTPD_DEFAULT_CONFIG (*C macro*), 221
httpd_err_code_t (*C++ enum*), 224
httpd_err_code_t::HTTPD_400_BAD_REQUEST (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_401_UNAUTHORIZED (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_403_FORBIDDEN (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_404_NOT_FOUND (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_405_METHOD_NOT_ALLOWED (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_408_REQ_TIMEOUT (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_411_LENGTH_REQUIRED (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_414_URI_TOO_LONG (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_431_REQ_HDR_FIELDS_TOO_LARGE (*C++ enumerator*), 225
httpd_err_code_t::HTTPD_500_INTERNAL_SERVER_ERROR (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_501_METHOD_NOT_IMPLEMENTED (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_505_VERSION_NOT_SUPPORTED (*C++ enumerator*), 224
httpd_err_code_t::HTTPD_ERR_CODE_MAX (*C++ enumerator*), 225
httpd_err_handler_func_t (*C++ type*), 222
httpd_free_ctx_fn_t (*C++ type*), 223
httpd_get_client_list (*C++ function*), 215
httpd_get_global_transport_ctx (*C++ function*), 215
httpd_get_global_user_ctx (*C++ function*), 215
httpd_handle_t (*C++ type*), 223
HTTPD_MAX_REQ_HDR_LEN (*C macro*), 220
HTTPD_MAX_URI_LEN (*C macro*), 220
httpd_method_t (*C++ type*), 223
httpd_open_func_t (*C++ type*), 223
httpd_pending_func_t (*C++ type*), 222
httpd_query_key_value (*C++ function*), 205
httpd_queue_work (*C++ function*), 213
httpd_recv_func_t (*C++ type*), 222
httpd_register_err_handler (*C++ function*), 212
httpd_register_uri_handler (*C++ function*), 201
httpd_req (*C++ struct*), 218
httpd_req::aux (*C++ member*), 219
httpd_req::content_len (*C++ member*), 219
httpd_req::free_ctx (*C++ member*), 219
httpd_req::handle (*C++ member*), 218
httpd_req::ignore_sess_ctx_changes (*C++ member*), 219
httpd_req::method (*C++ member*), 219
httpd_req::sess_ctx (*C++ member*), 219
httpd_req::uri (*C++ member*), 219
httpd_req::user_ctx (*C++ member*), 219
httpd_req_get_cookie_val (*C++ function*), 206
httpd_req_get_hdr_value_len (*C++ function*), 204
httpd_req_get_hdr_value_str (*C++ function*), 204
httpd_req_get_url_query_len (*C++ function*), 205
httpd_req_get_url_query_str (*C++ function*), 205
httpd_req_recv (*C++ function*), 203
httpd_req_t (*C++ type*), 221
httpd_req_to_sockfd (*C++ function*), 203
httpd_resp_send (*C++ function*), 207
httpd_resp_send_404 (*C++ function*), 210
httpd_resp_send_408 (*C++ function*), 210
httpd_resp_send_500 (*C++ function*), 210
httpd_resp_send_chunk (*C++ function*), 207
httpd_resp_send_err (*C++ function*), 209
httpd_resp_sendstr (*C++ function*), 208
httpd_resp_sendstr_chunk (*C++ function*), 208
httpd_resp_set_hdr (*C++ function*), 209
httpd_resp_set_status (*C++ function*), 208
httpd_resp_set_type (*C++ function*), 209
HTTPD_RESP_USE_STRLEN (*C macro*), 221
httpd_send (*C++ function*), 211
httpd_send_func_t (*C++ type*), 221
httpd_sess_get_ctx (*C++ function*), 214
httpd_sess_get_transport_ctx (*C++ function*), 214
httpd_sess_set_ctx (*C++ function*), 214
httpd_sess_set_pending_override (*C++ function*), 203
httpd_sess_set_recv_override (*C++ function*), 202
httpd_sess_set_send_override (*C++ function*), 203
httpd_sess_set_transport_ctx (*C++ function*), 214
httpd_sess_trigger_close (*C++ function*), 215
httpd_sess_update_lru_counter (*C++ function*), 215
HTTPD_SOCK_ERR_FAIL (*C macro*), 220

HTTPD_SOCK_ERR_INVALID (*C macro*), 220
HTTPD_SOCK_ERR_TIMEOUT (*C macro*), 220
httpd_socket_recv (*C++ function*), 212
httpd_socket_send (*C++ function*), 211
httpd_ssl_config (*C++ struct*), 227
httpd_ssl_config::cacert_len (*C++ member*), 227
httpd_ssl_config::cacert_pem (*C++ member*), 227
httpd_ssl_config::cert_select_cb (*C++ member*), 228
httpd_ssl_config::httpd (*C++ member*), 227
httpd_ssl_config::port_insecure (*C++ member*), 227
httpd_ssl_config::port_secure (*C++ member*), 227
httpd_ssl_config::prvtkey_len (*C++ member*), 227
httpd_ssl_config::prvtkey_pem (*C++ member*), 227
httpd_ssl_config::servercert (*C++ member*), 227
httpd_ssl_config::servercert_len (*C++ member*), 227
httpd_ssl_config::session_tickets (*C++ member*), 228
httpd_ssl_config::ssl_userdata (*C++ member*), 228
httpd_ssl_config::transport_mode (*C++ member*), 227
httpd_ssl_config::use_secure_element (*C++ member*), 228
httpd_ssl_config::user_cb (*C++ member*), 228
HTTPD_SSL_CONFIG_DEFAULT (*C macro*), 228
httpd_ssl_config_t (*C++ type*), 228
httpd_ssl_start (*C++ function*), 226
httpd_ssl_stop (*C++ function*), 226
httpd_ssl_transport_mode_t (*C++ enum*), 228
httpd_ssl_transport_mode_t::HTTPD_SSL_TRANSPORT_INSECURE (*C++ enumerator*), 228
httpd_ssl_transport_mode_t::HTTPD_SSL_TRANSPORT_SECURE (*C++ enumerator*), 228
httpd_ssl_user_cb_state_t (*C++ enum*), 228
httpd_ssl_user_cb_state_t::HTTPD_SSL_USER_CB_SESS_CLOSE (*C++ enumerator*), 229
httpd_ssl_user_cb_state_t::HTTPD_SSL_USER_CB_SESS_CREATE (*C++ enumerator*), 229
httpd_start (*C++ function*), 212
httpd_stop (*C++ function*), 213
HTTPD_TYPE_JSON (*C macro*), 220
HTTPD_TYPE_OCTET (*C macro*), 220
HTTPD_TYPE_TEXT (*C macro*), 220
httpd_unregister_uri (*C++ function*), 202
httpd_unregister_uri_handler (*C++ function*), 202
httpd_uri (*C++ struct*), 219
httpd_uri::handler (*C++ member*), 220
httpd_uri::method (*C++ member*), 219
httpd_uri::uri (*C++ member*), 219
httpd_uri::user_ctx (*C++ member*), 220
httpd_uri_match_func_t (*C++ type*), 223
httpd_uri_match_wildcard (*C++ function*), 206
httpd_uri_t (*C++ type*), 221
httpd_work_fn_t (*C++ type*), 224
HttpStatus_Code (*C++ enum*), 169
HttpStatus_Code::HttpStatus_BadRequest (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_Forbidden (*C++ enumerator*), 170
HttpStatus_Code::HttpStatus_Found (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_InternalError (*C++ enumerator*), 170
HttpStatus_Code::HttpStatus_MovedPermanently (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_MultipleChoices (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_NotFound (*C++ enumerator*), 170
HttpStatus_Code::HttpStatus_Ok (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_PermanentRedirect (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_SeeOther (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_TemporaryRedirect (*C++ enumerator*), 169
HttpStatus_Code::HttpStatus_Unauthorized (*C++ enumerator*), 169

I

i2c_ack_type_t (*C++ enum*), 1100
i2c_ack_type_t::I2C_MASTER_ACK (*C++ enumerator*), 1100
i2c_ack_type_t::I2C_MASTER_ACK_MAX (*C++ enumerator*), 1100
i2c_ack_type_t::I2C_MASTER_LAST_NACK (*C++ enumerator*), 1100
i2c_ack_type_t::I2C_MASTER_NACK (*C++ enumerator*), 1100
i2c_addr_mode_t (*C++ enum*), 1100
i2c_addr_mode_t::I2C_ADDR_BIT_10 (*C++ enumerator*), 1100
i2c_addr_mode_t::I2C_ADDR_BIT_7 (*C++ enumerator*), 1100
i2c_addr_mode_t::I2C_ADDR_BIT_MAX (*C++ enumerator*), 1100
I2C_APB_CLK_FREQ (*C macro*), 1098
i2c_clock_source_t (*C++ type*), 1099
i2c_cmd_handle_t (*C++ type*), 1099
i2c_cmd_link_create (*C++ function*), 1092
i2c_cmd_link_create_static (*C++ function*), 1091

i2c_cmd_link_delete (*C++ function*), 1092
i2c_cmd_link_delete_static (*C++ function*), 1092
i2c_config_t (*C++ struct*), 1097
i2c_config_t::addr_10bit_en (*C++ member*), 1098
i2c_config_t::clk_flags (*C++ member*), 1098
i2c_config_t::clk_speed (*C++ member*), 1097
i2c_config_t::master (*C++ member*), 1097
i2c_config_t::maximum_speed (*C++ member*), 1098
i2c_config_t::mode (*C++ member*), 1097
i2c_config_t::scl_io_num (*C++ member*), 1097
i2c_config_t::scl_pullup_en (*C++ member*), 1097
i2c_config_t::sda_io_num (*C++ member*), 1097
i2c_config_t::sda_pullup_en (*C++ member*), 1097
i2c_config_t::slave (*C++ member*), 1098
i2c_config_t::slave_addr (*C++ member*), 1098
i2c_driver_delete (*C++ function*), 1089
i2c_driver_install (*C++ function*), 1089
i2c_filter_disable (*C++ function*), 1095
i2c_filter_enable (*C++ function*), 1095
i2c_get_data_mode (*C++ function*), 1097
i2c_get_data_timing (*C++ function*), 1096
i2c_get_period (*C++ function*), 1094
i2c_get_start_timing (*C++ function*), 1095
i2c_get_stop_timing (*C++ function*), 1096
i2c_get_timeout (*C++ function*), 1096
I2C_INTERNAL_STRUCT_SIZE (*C macro*), 1098
I2C_LINK_RECOMMENDED_SIZE (*C macro*), 1098
i2c_master_cmd_begin (*C++ function*), 1093
i2c_master_read (*C++ function*), 1093
i2c_master_read_byte (*C++ function*), 1093
i2c_master_read_from_device (*C++ function*), 1091
i2c_master_start (*C++ function*), 1092
i2c_master_stop (*C++ function*), 1093
i2c_master_write (*C++ function*), 1092
i2c_master_write_byte (*C++ function*), 1092
i2c_master_write_read_device (*C++ function*), 1091
i2c_master_write_to_device (*C++ function*), 1090
i2c_mode_t (*C++ enum*), 1099
i2c_mode_t::I2C_MODE_MASTER (*C++ enumerator*), 1099
i2c_mode_t::I2C_MODE_MAX (*C++ enumerator*), 1099
i2c_mode_t::I2C_MODE_SLAVE (*C++ enumerator*), 1099
I2C_NUM_0 (*C macro*), 1098
I2C_NUM_1 (*C macro*), 1098
I2C_NUM_MAX (*C macro*), 1098
i2c_param_config (*C++ function*), 1090
i2c_port_t (*C++ type*), 1099
i2c_reset_rx_fifo (*C++ function*), 1090
i2c_reset_tx_fifo (*C++ function*), 1090
i2c_rw_t (*C++ enum*), 1099
i2c_rw_t::I2C_MASTER_READ (*C++ enumerator*), 1099
i2c_rw_t::I2C_MASTER_WRITE (*C++ enumerator*), 1099
I2C_SCLK_SRC_FLAG_AWARE_DFS (*C macro*), 1098
I2C_SCLK_SRC_FLAG_FOR_NOMAL (*C macro*), 1098
I2C_SCLK_SRC_FLAG_LIGHT_SLEEP (*C macro*), 1098
i2c_set_data_mode (*C++ function*), 1097
i2c_set_data_timing (*C++ function*), 1096
i2c_set_period (*C++ function*), 1094
i2c_set_pin (*C++ function*), 1090
i2c_set_start_timing (*C++ function*), 1095
i2c_set_stop_timing (*C++ function*), 1095
i2c_set_timeout (*C++ function*), 1096
i2c_slave_read_buffer (*C++ function*), 1094
i2c_slave_write_buffer (*C++ function*), 1094
i2c_trans_mode_t (*C++ enum*), 1099
i2c_trans_mode_t::I2C_DATA_MODE_LSB_FIRST (*C++ enumerator*), 1099
i2c_trans_mode_t::I2C_DATA_MODE_MAX (*C++ enumerator*), 1099
i2c_trans_mode_t::I2C_DATA_MODE_MSB_FIRST (*C++ enumerator*), 1099
i2s_chan_config_t (*C++ struct*), 1129
i2s_chan_config_t::auto_clear (*C++ member*), 1129
i2s_chan_config_t::dma_desc_num (*C++ member*), 1129
i2s_chan_config_t::dma_frame_num (*C++ member*), 1129
i2s_chan_config_t::id (*C++ member*), 1129
i2s_chan_config_t::role (*C++ member*), 1129
i2s_chan_handle_t (*C++ type*), 1131
i2s_chan_info_t (*C++ struct*), 1129
i2s_chan_info_t::dir (*C++ member*), 1130
i2s_chan_info_t::id (*C++ member*), 1130
i2s_chan_info_t::mode (*C++ member*), 1130
i2s_chan_info_t::pair_chan (*C++ member*), 1130
i2s_chan_info_t::role (*C++ member*), 1130
I2S_CHANNEL_DEFAULT_CONFIG (*C macro*), 1130
i2s_channel_disable (*C++ function*), 1127
i2s_channel_enable (*C++ function*), 1127
i2s_channel_get_info (*C++ function*), 1126
i2s_channel_init_pdm_rx_mode (*C++ function*), 1118

i2s_channel_init_pdm_tx_mode (*C++ function*), 1120
i2s_channel_init_std_mode (*C++ function*), 1114
i2s_channel_read (*C++ function*), 1128
i2s_channel_reconfig_pdm_rx_clock (*C++ function*), 1119
i2s_channel_reconfig_pdm_rx_gpio (*C++ function*), 1119
i2s_channel_reconfig_pdm_rx_slot (*C++ function*), 1119
i2s_channel_reconfig_pdm_tx_clock (*C++ function*), 1120
i2s_channel_reconfig_pdm_tx_gpio (*C++ function*), 1121
i2s_channel_reconfig_pdm_tx_slot (*C++ function*), 1121
i2s_channel_reconfig_std_clock (*C++ function*), 1114
i2s_channel_reconfig_std_gpio (*C++ function*), 1115
i2s_channel_reconfig_std_slot (*C++ function*), 1115
i2s_channel_register_event_callback (*C++ function*), 1128
i2s_channel_write (*C++ function*), 1127
i2s_clock_src_t (*C++ type*), 1132
i2s_comm_mode_t (*C++ enum*), 1131
i2s_comm_mode_t::I2S_COMM_MODE_NONE (*C++ enumerator*), 1131
i2s_comm_mode_t::I2S_COMM_MODE_PDM (*C++ enumerator*), 1131
i2s_comm_mode_t::I2S_COMM_MODE_STD (*C++ enumerator*), 1131
i2s_data_bit_width_t (*C++ enum*), 1132
i2s_data_bit_width_t::I2S_DATA_BIT_WIDTH_16BIT (*C++ enumerator*), 1133
i2s_data_bit_width_t::I2S_DATA_BIT_WIDTH_24BIT (*C++ enumerator*), 1133
i2s_data_bit_width_t::I2S_DATA_BIT_WIDTH_32BIT (*C++ enumerator*), 1133
i2s_data_bit_width_t::I2S_DATA_BIT_WIDTH_8BIT (*C++ enumerator*), 1133
i2s_del_channel (*C++ function*), 1126
i2s_dir_t (*C++ enum*), 1132
i2s_dir_t::I2S_DIR_RX (*C++ enumerator*), 1132
i2s_dir_t::I2S_DIR_TX (*C++ enumerator*), 1132
i2s_event_callbacks_t (*C++ struct*), 1129
i2s_event_callbacks_t::on_recv (*C++ member*), 1129
i2s_event_callbacks_t::on_recv_q_ovf (*C++ member*), 1129
i2s_event_callbacks_t::on_send_q_ovf (*C++ member*), 1129
i2s_event_callbacks_t::on_sent (*C++ member*), 1129
i2s_event_data_t (*C++ struct*), 1130
i2s_event_data_t::data (*C++ member*), 1130
i2s_event_data_t::size (*C++ member*), 1130
I2S_GPIO_UNUSED (*C macro*), 1130
i2s_isr_callback_t (*C++ type*), 1131
i2s_mclk_multiple_t (*C++ enum*), 1131
i2s_mclk_multiple_t::I2S_MCLK_MULTIPLE_128 (*C++ enumerator*), 1131
i2s_mclk_multiple_t::I2S_MCLK_MULTIPLE_256 (*C++ enumerator*), 1131
i2s_mclk_multiple_t::I2S_MCLK_MULTIPLE_384 (*C++ enumerator*), 1131
i2s_mclk_multiple_t::I2S_MCLK_MULTIPLE_512 (*C++ enumerator*), 1132
i2s_new_channel (*C++ function*), 1125
i2s_pdm_dsr_t (*C++ enum*), 1133
i2s_pdm_dsr_t::I2S_PDM_DSR_16S (*C++ enumerator*), 1133
i2s_pdm_dsr_t::I2S_PDM_DSR_8S (*C++ enumerator*), 1133
i2s_pdm_dsr_t::I2S_PDM_DSR_MAX (*C++ enumerator*), 1133
i2s_pdm_rx_clk_config_t (*C++ struct*), 1122
i2s_pdm_rx_clk_config_t::clk_src (*C++ member*), 1122
i2s_pdm_rx_clk_config_t::dn_sample_mode (*C++ member*), 1122
i2s_pdm_rx_clk_config_t::mclk_multiple (*C++ member*), 1122
i2s_pdm_rx_clk_config_t::sample_rate_hz (*C++ member*), 1122
I2S_PDM_RX_CLK_DEFAULT_CONFIG (*C macro*), 1125
i2s_pdm_rx_config_t (*C++ struct*), 1122
i2s_pdm_rx_config_t::clk_cfg (*C++ member*), 1123
i2s_pdm_rx_config_t::gpio_cfg (*C++ member*), 1123
i2s_pdm_rx_config_t::slot_cfg (*C++ member*), 1123
i2s_pdm_rx_gpio_config_t (*C++ struct*), 1122
i2s_pdm_rx_gpio_config_t::clk (*C++ member*), 1122
i2s_pdm_rx_gpio_config_t::clk_inv (*C++ member*), 1122
i2s_pdm_rx_gpio_config_t::din (*C++ member*), 1122
i2s_pdm_rx_gpio_config_t::invert_flags (*C++ member*), 1122
i2s_pdm_rx_slot_config_t (*C++ struct*), 1121
i2s_pdm_rx_slot_config_t::data_bit_width (*C++ member*), 1122
i2s_pdm_rx_slot_config_t::slot_bit_width (*C++ member*), 1122
i2s_pdm_rx_slot_config_t::slot_mask (*C++ member*), 1122
i2s_pdm_rx_slot_config_t::slot_mode (*C++ member*), 1122

I2S_PDM_RX_SLOT_DEFAULT_CONFIG (*C macro*), 1125
i2s_pdm_sig_scale_t (*C++ enum*), 1133
i2s_pdm_sig_scale_t::I2S_PDM_SIG_SCALING_DIV_2 (*C++ enumerator*), 1134
i2s_pdm_sig_scale_t::I2S_PDM_SIG_SCALING_MUL_1 (*C++ enumerator*), 1134
i2s_pdm_sig_scale_t::I2S_PDM_SIG_SCALING_MUL_2 (*C++ enumerator*), 1134
i2s_pdm_sig_scale_t::I2S_PDM_SIG_SCALING_MUL_4 (*C++ enumerator*), 1134
i2s_pdm_slot_mask_t (*C++ enum*), 1134
i2s_pdm_slot_mask_t::I2S_PDM_SLOT_BOTH (*C++ enumerator*), 1134
i2s_pdm_slot_mask_t::I2S_PDM_SLOT_LEFT (*C++ enumerator*), 1134
i2s_pdm_slot_mask_t::I2S_PDM_SLOT_RIGHT (*C++ enumerator*), 1134
i2s_pdm_tx_clk_config_t (*C++ struct*), 1123
i2s_pdm_tx_clk_config_t::clk_src (*C++ member*), 1124
i2s_pdm_tx_clk_config_t::mclk_multiple (*C++ member*), 1124
i2s_pdm_tx_clk_config_t::sample_rate_hz (*C++ member*), 1124
i2s_pdm_tx_clk_config_t::up_sample_fp (*C++ member*), 1124
i2s_pdm_tx_clk_config_t::up_sample_fs (*C++ member*), 1124
I2S_PDM_TX_CLK_DEFAULT_CONFIG (*C macro*), 1125
i2s_pdm_tx_config_t (*C++ struct*), 1124
i2s_pdm_tx_config_t::clk_cfg (*C++ member*), 1124
i2s_pdm_tx_config_t::gpio_cfg (*C++ member*), 1125
i2s_pdm_tx_config_t::slot_cfg (*C++ member*), 1124
i2s_pdm_tx_gpio_config_t (*C++ struct*), 1124
i2s_pdm_tx_gpio_config_t::clk (*C++ member*), 1124
i2s_pdm_tx_gpio_config_t::clk_inv (*C++ member*), 1124
i2s_pdm_tx_gpio_config_t::dout (*C++ member*), 1124
i2s_pdm_tx_gpio_config_t::invert_flags (*C++ member*), 1124
i2s_pdm_tx_slot_config_t (*C++ struct*), 1123
i2s_pdm_tx_slot_config_t::data_bit_width (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::hp_scale (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::lp_scale (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::sd_prescale (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::sd_scale (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::sinc_scale (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::slot_bit_width (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::slot_mask (*C++ member*), 1123
i2s_pdm_tx_slot_config_t::slot_mode (*C++ member*), 1123
I2S_PDM_TX_SLOT_DEFAULT_CONFIG (*C macro*), 1125
i2s_port_t (*C++ enum*), 1131
i2s_port_t::I2S_NUM_0 (*C++ enumerator*), 1131
i2s_port_t::I2S_NUM_1 (*C++ enumerator*), 1131
i2s_port_t::I2S_NUM_AUTO (*C++ enumerator*), 1131
i2s_role_t (*C++ enum*), 1132
i2s_role_t::I2S_ROLE_MASTER (*C++ enumerator*), 1132
i2s_role_t::I2S_ROLE_SLAVE (*C++ enumerator*), 1132
i2s_slot_bit_width_t (*C++ enum*), 1133
i2s_slot_bit_width_t::I2S_SLOT_BIT_WIDTH_16BIT (*C++ enumerator*), 1133
i2s_slot_bit_width_t::I2S_SLOT_BIT_WIDTH_24BIT (*C++ enumerator*), 1133
i2s_slot_bit_width_t::I2S_SLOT_BIT_WIDTH_32BIT (*C++ enumerator*), 1133
i2s_slot_bit_width_t::I2S_SLOT_BIT_WIDTH_8BIT (*C++ enumerator*), 1133
i2s_slot_bit_width_t::I2S_SLOT_BIT_WIDTH_AUTO (*C++ enumerator*), 1133
i2s_slot_mode_t (*C++ enum*), 1132
i2s_slot_mode_t::I2S_SLOT_MODE_MONO (*C++ enumerator*), 1132
i2s_slot_mode_t::I2S_SLOT_MODE_STEREO (*C++ enumerator*), 1132
i2s_std_clk_config_t (*C++ struct*), 1116
i2s_std_clk_config_t::clk_src (*C++ member*), 1116
i2s_std_clk_config_t::mclk_multiple (*C++ member*), 1116
i2s_std_clk_config_t::sample_rate_hz (*C++ member*), 1116
I2S_STD_CLK_DEFAULT_CONFIG (*C macro*), 1118
i2s_std_config_t (*C++ struct*), 1117
i2s_std_config_t::clk_cfg (*C++ member*), 1117
i2s_std_config_t::gpio_cfg (*C++ member*), 1117
i2s_std_config_t::slot_cfg (*C++ member*), 1117
i2s_std_gpio_config_t (*C++ struct*), 1116
i2s_std_gpio_config_t::bclk (*C++ member*), 1117
i2s_std_gpio_config_t::bclk_inv (*C++ member*), 1117

i2s_std_gpio_config_t::din (C++ member), 1117
i2s_std_gpio_config_t::dout (C++ member), 1117
i2s_std_gpio_config_t::invert_flags (C++ member), 1117
i2s_std_gpio_config_t::mclk (C++ member), 1117
i2s_std_gpio_config_t::mclk_inv (C++ member), 1117
i2s_std_gpio_config_t::ws (C++ member), 1117
i2s_std_gpio_config_t::ws_inv (C++ member), 1117
I2S_STD_MSB_SLOT_DEFAULT_CONFIG (C macro), 1118
I2S_STD_PCM_SLOT_DEFAULT_CONFIG (C macro), 1118
I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG (C macro), 1117
i2s_std_slot_config_t (C++ struct), 1115
i2s_std_slot_config_t::bit_shift (C++ member), 1116
i2s_std_slot_config_t::data_bit_width (C++ member), 1116
i2s_std_slot_config_t::msb_right (C++ member), 1116
i2s_std_slot_config_t::slot_bit_width (C++ member), 1116
i2s_std_slot_config_t::slot_mask (C++ member), 1116
i2s_std_slot_config_t::slot_mode (C++ member), 1116
i2s_std_slot_config_t::ws_pol (C++ member), 1116
i2s_std_slot_config_t::ws_width (C++ member), 1116
i2s_std_slot_mask_t (C++ enum), 1134
i2s_std_slot_mask_t::I2S_STD_SLOT_BOTH (C++ enumerator), 1134
i2s_std_slot_mask_t::I2S_STD_SLOT_LEFT (C++ enumerator), 1134
i2s_std_slot_mask_t::I2S_STD_SLOT_RIGHT (C++ enumerator), 1134
I_ADC (C macro), 2175
I_ADDI (C macro), 2176
I_ADDR (C macro), 2176
I_ANDI (C macro), 2176
I_ANDR (C macro), 2176
I_BGE (C macro), 2175
I_BL (C macro), 2175
I_BXFI (C macro), 2176
I_BXFR (C macro), 2176
I_BXI (C macro), 2176
I_BXR (C macro), 2176
I_BXZI (C macro), 2176
I_BXZR (C macro), 2176
I_DELAY (C macro), 2174
I_END (C macro), 2175
I_HALT (C macro), 2174
I_I2C_READ (C macro), 2178
I_I2C_RW (C macro), 2178
I_I2C_WRITE (C macro), 2178
I_JUMPS (C macro), 2178
I_LD (C macro), 2175
I_LSHI (C macro), 2176
I_LSHR (C macro), 2176
I_MOVI (C macro), 2176
I_MOVR (C macro), 2176
I_ORI (C macro), 2176
I_ORR (C macro), 2176
I_RD_REG (C macro), 2174
I_RSHI (C macro), 2177
I_RSHR (C macro), 2176
I_SLEEP_CYCLE_SEL (C macro), 2175
I_ST (C macro), 2175
I_STAGE_DEC (C macro), 2177
I_STAGE_INC (C macro), 2177
I_STAGE_RST (C macro), 2178
I_SUBI (C macro), 2176
I_SUBR (C macro), 2176
I_TSENS (C macro), 2175
I_WAKE (C macro), 2174
I_WR_REG (C macro), 2174
I_WR_REG_BIT (C macro), 2174
intr_handle_data_t (C++ type), 2073
intr_handle_t (C++ type), 2074
intr_handler_t (C++ type), 2073
IP2STR (C macro), 1015
ip_event_add_ip6_t (C++ struct), 1006
ip_event_add_ip6_t::addr (C++ member), 1006
ip_event_add_ip6_t::preferred (C++ member), 1006
ip_event_ap_staipassigned_t (C++ struct), 1006
ip_event_ap_staipassigned_t::esp_netif (C++ member), 1007
ip_event_ap_staipassigned_t::ip (C++ member), 1007
ip_event_ap_staipassigned_t::mac (C++ member), 1007
ip_event_got_ip6_t (C++ struct), 1006
ip_event_got_ip6_t::esp_netif (C++ member), 1006
ip_event_got_ip6_t::ip6_info (C++ member), 1006
ip_event_got_ip6_t::ip_index (C++ member), 1006
ip_event_got_ip_t (C++ struct), 1006
ip_event_got_ip_t::esp_netif (C++ member), 1006
ip_event_got_ip_t::ip_changed (C++ member), 1006
ip_event_got_ip_t::ip_info (C++ member), 1006

ip_event_t (*C++ enum*), 1012
ip_event_t::IP_EVENT_AP_STAIPASSIGNED (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_ETH_GOT_IP (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_ETH_LOST_IP (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_GOT_IP6 (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_PPP_GOT_IP (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_PPP_LOST_IP (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_STA_GOT_IP (*C++ enumerator*), 1012
ip_event_t::IP_EVENT_STA_LOST_IP (*C++ enumerator*), 1012
IPSTR (*C macro*), 1014
IPV62STR (*C macro*), 1015
IPV6STR (*C macro*), 1015

J

JUMPS_GE (*C macro*), 2173
JUMPS_LE (*C macro*), 2173
JUMPS_LT (*C macro*), 2173

L

l2tap_ioctl_opt_t (*C++ enum*), 1017
l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL (*C++ enumerator*), 1017
l2tap_ioctl_opt_t::L2TAP_G_INTF_DEVICE (*C++ enumerator*), 1017
l2tap_ioctl_opt_t::L2TAP_G_RCV_FILTER (*C++ enumerator*), 1017
l2tap_ioctl_opt_t::L2TAP_S_DEVICE_DRV_HNDL (*C++ enumerator*), 1017
l2tap_ioctl_opt_t::L2TAP_S_INTF_DEVICE (*C++ enumerator*), 1017
l2tap_ioctl_opt_t::L2TAP_S_RCV_FILTER (*C++ enumerator*), 1017
l2tap_iodriver_handle (*C++ type*), 1017
L2TAP_VFS_CONFIG_DEFAULT (*C macro*), 1016
l2tap_vfs_config_t (*C++ struct*), 1016
l2tap_vfs_config_t::base_path (*C++ member*), 1016
L2TAP_VFS_DEFAULT_PATH (*C macro*), 1016
lcd_clock_source_t (*C++ type*), 1139
lcd_color_range_t (*C++ enum*), 1140
lcd_color_range_t::LCD_COLOR_RANGE_FULL (*C++ enumerator*), 1140
lcd_color_range_t::LCD_COLOR_RANGE_LIMIT (*C++ enumerator*), 1140
lcd_color_rgb_endian_t (*C++ type*), 1139
lcd_color_space_t (*C++ enum*), 1140
lcd_color_space_t::LCD_COLOR_SPACE_RGB (*C++ enumerator*), 1140
lcd_color_space_t::LCD_COLOR_SPACE_YUV (*C++ enumerator*), 1140
lcd_rgb_data_endian_t (*C++ enum*), 1140
lcd_rgb_data_endian_t::LCD_RGB_DATA_ENDIAN_BIG (*C++ enumerator*), 1140
lcd_rgb_data_endian_t::LCD_RGB_DATA_ENDIAN_LITTLE (*C++ enumerator*), 1140
lcd_rgb_element_order_t (*C++ enum*), 1140
lcd_rgb_element_order_t::LCD_RGB_ELEMENT_ORDER_BG (*C++ enumerator*), 1140
lcd_rgb_element_order_t::LCD_RGB_ELEMENT_ORDER_RG (*C++ enumerator*), 1140
LCD_RGB_ENDIAN_BGR (*C macro*), 1139
LCD_RGB_ENDIAN_RGB (*C macro*), 1139
lcd_yuv_conv_std_t (*C++ enum*), 1141
lcd_yuv_conv_std_t::LCD_YUV_CONV_STD_BT601 (*C++ enumerator*), 1141
lcd_yuv_conv_std_t::LCD_YUV_CONV_STD_BT709 (*C++ enumerator*), 1141
lcd_yuv_sample_t (*C++ enum*), 1140
lcd_yuv_sample_t::LCD_YUV_SAMPLE_411 (*C++ enumerator*), 1141
lcd_yuv_sample_t::LCD_YUV_SAMPLE_420 (*C++ enumerator*), 1140
lcd_yuv_sample_t::LCD_YUV_SAMPLE_422 (*C++ enumerator*), 1140
LEDC_APB_CLK_HZ (*C macro*), 1166
ledc_bind_channel_timer (*C++ function*), 1160
ledc_cb_event_t (*C++ enum*), 1166
ledc_cb_event_t::LEDC_FADE_END_EVT (*C++ enumerator*), 1167
ledc_cb_param_t (*C++ struct*), 1165
ledc_cb_param_t::channel (*C++ member*), 1165
ledc_cb_param_t::duty (*C++ member*), 1166
ledc_cb_param_t::event (*C++ member*), 1165
ledc_cb_param_t::speed_mode (*C++ member*), 1165
ledc_cb_register (*C++ function*), 1164
ledc_cb_t (*C++ type*), 1166
ledc_cbs_t (*C++ struct*), 1166
ledc_cbs_t::fade_cb (*C++ member*), 1166
ledc_channel_config (*C++ function*), 1156
ledc_channel_config_t (*C++ struct*), 1164
ledc_channel_config_t::channel (*C++ member*), 1164
ledc_channel_config_t::duty (*C++ member*), 1164
ledc_channel_config_t::flags (*C++ member*), 1165
ledc_channel_config_t::gpio_num (*C++ member*), 1164
ledc_channel_config_t::hpoint (*C++ member*), 1165
ledc_channel_config_t::intr_type (*C++ member*), 1164
ledc_channel_config_t::output_invert (*C++ member*), 1165
ledc_channel_config_t::speed_mode

(C++ *member*), 1164
ledc_channel_config_t::timer_sel (C++ *member*), 1164
ledc_channel_t (C++ *enum*), 1169
ledc_channel_t::LEDC_CHANNEL_0 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_1 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_2 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_3 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_4 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_5 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_6 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_7 (C++ *enumerator*), 1169
ledc_channel_t::LEDC_CHANNEL_MAX (C++ *enumerator*), 1169
ledc_clk_cfg_t (C++ *enum*), 1168
ledc_clk_cfg_t::LEDC_AUTO_CLK (C++ *enumerator*), 1168
ledc_clk_cfg_t::LEDC_USE_APB_CLK (C++ *enumerator*), 1168
ledc_clk_cfg_t::LEDC_USE_REF_TICK (C++ *enumerator*), 1168
ledc_clk_cfg_t::LEDC_USE_RTC8M_CLK (C++ *enumerator*), 1168
ledc_clk_src_t (C++ *enum*), 1168
ledc_clk_src_t::LEDC_APB_CLK (C++ *enumerator*), 1168
ledc_clk_src_t::LEDC_REF_TICK (C++ *enumerator*), 1168
ledc_clk_src_t::LEDC_SCLK (C++ *enumerator*), 1168
ledc_duty_direction_t (C++ *enum*), 1167
ledc_duty_direction_t::LEDC_DUTY_DIR_DECREASE (C++ *enumerator*), 1167
ledc_duty_direction_t::LEDC_DUTY_DIR_INCREASE (C++ *enumerator*), 1167
ledc_duty_direction_t::LEDC_DUTY_DIR_MAX (C++ *enumerator*), 1167
LEDC_ERR_DUTY (C *macro*), 1166
LEDC_ERR_VAL (C *macro*), 1166
ledc_fade_func_install (C++ *function*), 1162
ledc_fade_func_uninstall (C++ *function*), 1162
ledc_fade_mode_t (C++ *enum*), 1171
ledc_fade_mode_t::LEDC_FADE_MAX (C++ *enumerator*), 1171
ledc_fade_mode_t::LEDC_FADE_NO_WAIT (C++ *enumerator*), 1171
ledc_fade_mode_t::LEDC_FADE_WAIT_DONE (C++ *enumerator*), 1171
ledc_fade_start (C++ *function*), 1162
ledc_get_duty (C++ *function*), 1158
ledc_get_freq (C++ *function*), 1157
ledc_get_hpoint (C++ *function*), 1158
ledc_intr_type_t (C++ *enum*), 1167
ledc_intr_type_t::LEDC_INTR_DISABLE (C++ *enumerator*), 1167
ledc_intr_type_t::LEDC_INTR_FADE_END (C++ *enumerator*), 1167
ledc_intr_type_t::LEDC_INTR_MAX (C++ *enumerator*), 1167
ledc_isr_handle_t (C++ *type*), 1166
ledc_isr_register (C++ *function*), 1159
ledc_mode_t (C++ *enum*), 1167
ledc_mode_t::LEDC_HIGH_SPEED_MODE (C++ *enumerator*), 1167
ledc_mode_t::LEDC_LOW_SPEED_MODE (C++ *enumerator*), 1167
ledc_mode_t::LEDC_SPEED_MODE_MAX (C++ *enumerator*), 1167
LEDC_REF_CLK_HZ (C *macro*), 1166
ledc_set_duty (C++ *function*), 1158
ledc_set_duty_and_update (C++ *function*), 1162
ledc_set_duty_with_hpoint (C++ *function*), 1157
ledc_set_fade (C++ *function*), 1159
ledc_set_fade_step_and_start (C++ *function*), 1163
ledc_set_fade_time_and_start (C++ *function*), 1163
ledc_set_fade_with_step (C++ *function*), 1160
ledc_set_fade_with_time (C++ *function*), 1161
ledc_set_freq (C++ *function*), 1157
ledc_set_pin (C++ *function*), 1156
ledc_slow_clk_sel_t (C++ *enum*), 1167
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_APB (C++ *enumerator*), 1168
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_RTC8M (C++ *enumerator*), 1167
ledc_stop (C++ *function*), 1157
ledc_timer_bit_t (C++ *enum*), 1169
ledc_timer_bit_t::LEDC_TIMER_10_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_11_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_12_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_13_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_14_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_15_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_16_BIT (C++ *enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_17_BIT

(*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_18_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_19_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_1_BIT (*C++ enumerator*), 1169
ledc_timer_bit_t::LEDC_TIMER_20_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_2_BIT (*C++ enumerator*), 1169
ledc_timer_bit_t::LEDC_TIMER_3_BIT (*C++ enumerator*), 1169
ledc_timer_bit_t::LEDC_TIMER_4_BIT (*C++ enumerator*), 1169
ledc_timer_bit_t::LEDC_TIMER_5_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_6_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_7_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_8_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_9_BIT (*C++ enumerator*), 1170
ledc_timer_bit_t::LEDC_TIMER_BIT_MAX (*C++ enumerator*), 1171
ledc_timer_config (*C++ function*), 1156
ledc_timer_config_t (*C++ struct*), 1165
ledc_timer_config_t::clk_cfg (*C++ member*), 1165
ledc_timer_config_t::duty_resolution (*C++ member*), 1165
ledc_timer_config_t::freq_hz (*C++ member*), 1165
ledc_timer_config_t::speed_mode (*C++ member*), 1165
ledc_timer_config_t::timer_num (*C++ member*), 1165
ledc_timer_pause (*C++ function*), 1160
ledc_timer_resume (*C++ function*), 1160
ledc_timer_rst (*C++ function*), 1160
ledc_timer_set (*C++ function*), 1159
ledc_timer_t (*C++ enum*), 1168
ledc_timer_t::LEDC_TIMER_0 (*C++ enumerator*), 1168
ledc_timer_t::LEDC_TIMER_1 (*C++ enumerator*), 1168
ledc_timer_t::LEDC_TIMER_2 (*C++ enumerator*), 1168
ledc_timer_t::LEDC_TIMER_3 (*C++ enumerator*), 1169
ledc_timer_t::LEDC_TIMER_MAX (*C++ enumerator*), 1169
ledc_update_duty (*C++ function*), 1156
linenoiseCompletions (*C++ type*), 1849

M

M_BGE (*C macro*), 2177
M_BL (*C macro*), 2177
M_BRANCH (*C macro*), 2177
M_BSEQ (*C macro*), 2178
M_BSGE (*C macro*), 2178
M_BSGT (*C macro*), 2178
M_BSLE (*C macro*), 2178
M_BSLT (*C macro*), 2178
M_BX (*C macro*), 2177
M_BXF (*C macro*), 2177
M_BXZ (*C macro*), 2177
M_LABEL (*C macro*), 2177
M_LABELPC (*C macro*), 2177
M_MOVL (*C macro*), 2177
MAC2STR (*C macro*), 2088
MACSTR (*C macro*), 2088
MALLOC_CAP_32BIT (*C macro*), 2038
MALLOC_CAP_8BIT (*C macro*), 2038
MALLOC_CAP_DEFAULT (*C macro*), 2038
MALLOC_CAP_DMA (*C macro*), 2038
MALLOC_CAP_EXEC (*C macro*), 2037
MALLOC_CAP_INTERNAL (*C macro*), 2038
MALLOC_CAP_INVALID (*C macro*), 2038
MALLOC_CAP_IRAM_8BIT (*C macro*), 2038
MALLOC_CAP_PID2 (*C macro*), 2038
MALLOC_CAP_PID3 (*C macro*), 2038
MALLOC_CAP_PID4 (*C macro*), 2038
MALLOC_CAP_PID5 (*C macro*), 2038
MALLOC_CAP_PID6 (*C macro*), 2038
MALLOC_CAP_PID7 (*C macro*), 2038
MALLOC_CAP_RETENTION (*C macro*), 2038
MALLOC_CAP_RTCRAM (*C macro*), 2038
MALLOC_CAP_SPIRAM (*C macro*), 2038
MAX_BLE_DEVNAME_LEN (*C macro*), 1689
MAX_BLE_MANUFACTURER_DATA_LEN (*C macro*), 1689
MAX_FDS (*C macro*), 1820
MAX_PASSPHRASE_LEN (*C macro*), 916
MAX_SSID_LEN (*C macro*), 916
MAX_WPS_AP_CRED (*C macro*), 916
mcpwm_brake_config_t (*C++ struct*), 1199
mcpwm_brake_config_t::brake_mode (*C++ member*), 1199
mcpwm_brake_config_t::cbc_recover_on_tep (*C++ member*), 1199
mcpwm_brake_config_t::cbc_recover_on_tez (*C++ member*), 1199
mcpwm_brake_config_t::fault (*C++ member*), 1199
mcpwm_brake_config_t::flags (*C++ member*), 1199
mcpwm_brake_event_cb_t (*C++ type*), 1220
mcpwm_brake_event_data_t (*C++ struct*), 1219
mcpwm_cap_channel_handle_t (*C++ type*), 1220
mcpwm_cap_timer_handle_t (*C++ type*), 1220

mcpwm_capture_channel_config_t (*C++ struct*), 1217
mcpwm_capture_channel_config_t::flags (*C++ member*), 1218
mcpwm_capture_channel_config_t::gpio_num (*C++ member*), 1217
mcpwm_capture_channel_config_t::intr_priority (*C++ member*), 1217
mcpwm_capture_channel_config_t::invert_cap_signal (*C++ member*), 1218
mcpwm_capture_channel_config_t::io_loop_back (*C++ member*), 1218
mcpwm_capture_channel_config_t::keep_io_conf_at_exit (*C++ member*), 1218
mcpwm_capture_channel_config_t::neg_edge (*C++ member*), 1218
mcpwm_capture_channel_config_t::pos_edge (*C++ member*), 1218
mcpwm_capture_channel_config_t::prescale (*C++ member*), 1217
mcpwm_capture_channel_config_t::pull_down (*C++ member*), 1218
mcpwm_capture_channel_config_t::pull_up (*C++ member*), 1218
mcpwm_capture_channel_disable (*C++ function*), 1216
mcpwm_capture_channel_enable (*C++ function*), 1215
mcpwm_capture_channel_register_event_callbacks (*C++ function*), 1216
mcpwm_capture_channel_trigger_soft_catch (*C++ function*), 1216
mcpwm_capture_clock_source_t (*C++ type*), 1221
mcpwm_capture_edge_t (*C++ enum*), 1223
mcpwm_capture_edge_t::MCPWM_CAP_EDGE_NEG (*C++ enumerator*), 1223
mcpwm_capture_edge_t::MCPWM_CAP_EDGE_POS (*C++ enumerator*), 1223
mcpwm_capture_event_callbacks_t (*C++ struct*), 1218
mcpwm_capture_event_callbacks_t::on_cap (*C++ member*), 1218
mcpwm_capture_event_cb_t (*C++ type*), 1220
mcpwm_capture_event_data_t (*C++ struct*), 1219
mcpwm_capture_event_data_t::cap_edge (*C++ member*), 1219
mcpwm_capture_event_data_t::cap_value (*C++ member*), 1219
mcpwm_capture_timer_config_t (*C++ struct*), 1217
mcpwm_capture_timer_config_t::clk_src (*C++ member*), 1217
mcpwm_capture_timer_config_t::group_id (*C++ member*), 1217
mcpwm_capture_timer_disable (*C++ function*), 1214
mcpwm_capture_timer_enable (*C++ function*), 1214
mcpwm_capture_timer_get_resolution (*C++ function*), 1214
mcpwm_capture_timer_set_phase_on_sync (*C++ function*), 1215
mcpwm_capture_timer_start (*C++ function*), 1214
mcpwm_capture_timer_stop (*C++ function*), 1214
mcpwm_capture_timer_sync_phase_config_t (*C++ struct*), 1217
mcpwm_capture_timer_sync_phase_config_t::count_value (*C++ member*), 1217
mcpwm_capture_timer_sync_phase_config_t::direction (*C++ member*), 1217
mcpwm_capture_timer_sync_phase_config_t::sync_src (*C++ member*), 1217
mcpwm_carrier_clock_source_t (*C++ type*), 1221
mcpwm_carrier_config_t (*C++ struct*), 1200
mcpwm_carrier_config_t::clk_src (*C++ member*), 1200
mcpwm_carrier_config_t::duty_cycle (*C++ member*), 1200
mcpwm_carrier_config_t::first_pulse_duration_us (*C++ member*), 1200
mcpwm_carrier_config_t::flags (*C++ member*), 1200
mcpwm_carrier_config_t::frequency_hz (*C++ member*), 1200
mcpwm_carrier_config_t::invert_after_modulate (*C++ member*), 1200
mcpwm_carrier_config_t::invert_before_modulate (*C++ member*), 1200
mcpwm_cmpr_handle_t (*C++ type*), 1219
mcpwm_comparator_config_t (*C++ struct*), 1201
mcpwm_comparator_config_t::flags (*C++ member*), 1202
mcpwm_comparator_config_t::intr_priority (*C++ member*), 1201
mcpwm_comparator_config_t::update_cmp_on_sync (*C++ member*), 1202
mcpwm_comparator_config_t::update_cmp_on_tep (*C++ member*), 1202
mcpwm_comparator_config_t::update_cmp_on_tez (*C++ member*), 1201
mcpwm_comparator_event_callbacks_t (*C++ struct*), 1202
mcpwm_comparator_event_callbacks_t::on_reach (*C++ member*), 1202
mcpwm_comparator_register_event_callbacks (*C++ function*), 1201
mcpwm_comparator_set_compare_value (*C++ function*), 1201
mcpwm_compare_event_cb_t (*C++ type*), 1220
mcpwm_compare_event_data_t (*C++ struct*),

1219
mcpwm_compare_event_data_t::compare_ticks (C++ member), 1219
mcpwm_compare_event_data_t::direction (C++ member), 1219
mcpwm_dead_time_config_t (C++ struct), 1208
mcpwm_dead_time_config_t::flags (C++ member), 1208
mcpwm_dead_time_config_t::invert_output (C++ member), 1208
mcpwm_dead_time_config_t::negedge_delay_ticks (C++ member), 1208
mcpwm_dead_time_config_t::posedge_delay_ticks (C++ member), 1208
mcpwm_del_capture_channel (C++ function), 1215
mcpwm_del_capture_timer (C++ function), 1213
mcpwm_del_comparator (C++ function), 1200
mcpwm_del_fault (C++ function), 1209
mcpwm_del_generator (C++ function), 1202
mcpwm_del_operator (C++ function), 1197
mcpwm_del_sync_src (C++ function), 1212
mcpwm_del_timer (C++ function), 1193
mcpwm_fault_event_callbacks_t (C++ struct), 1211
mcpwm_fault_event_callbacks_t::on_fault_exit (C++ member), 1211
mcpwm_fault_event_callbacks_t::on_fault_enter (C++ member), 1211
mcpwm_fault_event_cb_t (C++ type), 1220
mcpwm_fault_event_data_t (C++ struct), 1219
mcpwm_fault_handle_t (C++ type), 1220
mcpwm_fault_register_event_callbacks (C++ function), 1210
MCPWM_GEN_BRAKE_EVENT_ACTION (C macro), 1208
MCPWM_GEN_BRAKE_EVENT_ACTION_END (C macro), 1208
mcpwm_gen_brake_event_action_t (C++ struct), 1207
mcpwm_gen_brake_event_action_t::action (C++ member), 1207
mcpwm_gen_brake_event_action_t::brake_mode (C++ member), 1207
mcpwm_gen_brake_event_action_t::direction (C++ member), 1207
MCPWM_GEN_COMPARE_EVENT_ACTION (C macro), 1208
MCPWM_GEN_COMPARE_EVENT_ACTION_END (C macro), 1208
mcpwm_gen_compare_event_action_t (C++ struct), 1207
mcpwm_gen_compare_event_action_t::action (C++ member), 1207
mcpwm_gen_compare_event_action_t::comparator (C++ member), 1207
mcpwm_gen_compare_event_action_t::direction (C++ member), 1207
MCPWM_GEN_FAULT_EVENT_ACTION (C macro), 1209
mcpwm_gen_fault_event_action_t (C++ struct), 1207
mcpwm_gen_fault_event_action_t::action (C++ member), 1208
mcpwm_gen_fault_event_action_t::direction (C++ member), 1207
mcpwm_gen_fault_event_action_t::fault (C++ member), 1207
mcpwm_gen_handle_t (C++ type), 1219
MCPWM_GEN_SYNC_EVENT_ACTION (C macro), 1209
mcpwm_gen_sync_event_action_t (C++ struct), 1208
mcpwm_gen_sync_event_action_t::action (C++ member), 1208
mcpwm_gen_sync_event_action_t::direction (C++ member), 1208
mcpwm_gen_sync_event_action_t::sync (C++ member), 1208
MCPWM_GEN_TIMER_EVENT_ACTION (C macro), 1208
MCPWM_GEN_TIMER_EVENT_ACTION_END (C macro), 1208
mcpwm_gen_timer_event_action_t (C++ struct), 1206
mcpwm_gen_timer_event_action_t::action (C++ member), 1207
mcpwm_gen_timer_event_action_t::direction (C++ member), 1206
mcpwm_gen_timer_event_action_t::event (C++ member), 1207
mcpwm_generator_action_t (C++ enum), 1222
mcpwm_generator_action_t::MCPWM_GEN_ACTION_HIGH (C++ enumerator), 1222
mcpwm_generator_action_t::MCPWM_GEN_ACTION_KEEP (C++ enumerator), 1222
mcpwm_generator_action_t::MCPWM_GEN_ACTION_LOW (C++ enumerator), 1222
mcpwm_generator_action_t::MCPWM_GEN_ACTION_TOGGLE (C++ enumerator), 1222
mcpwm_generator_config_t (C++ struct), 1206
mcpwm_generator_config_t::flags (C++ member), 1206
mcpwm_generator_config_t::gen_gpio_num (C++ member), 1206
mcpwm_generator_config_t::invert_pwm (C++ member), 1206
mcpwm_generator_config_t::io_loop_back (C++ member), 1206
mcpwm_generator_config_t::io_od_mode (C++ member), 1206
mcpwm_generator_config_t::pull_down (C++ member), 1206
mcpwm_generator_config_t::pull_up (C++ member), 1206

mcpwm_generator_set_action_on_brake_event (C++ function), 1204
mcpwm_generator_set_action_on_compare_event (C++ function), 1204
mcpwm_generator_set_action_on_fault_event (C++ function), 1205
mcpwm_generator_set_action_on_sync_event (C++ function), 1205
mcpwm_generator_set_action_on_timer_event (C++ function), 1203
mcpwm_generator_set_actions_on_brake_event (C++ function), 1204
mcpwm_generator_set_actions_on_compare_event (C++ function), 1204
mcpwm_generator_set_actions_on_timer_event (C++ function), 1203
mcpwm_generator_set_dead_time (C++ function), 1205
mcpwm_generator_set_force_level (C++ function), 1202
mcpwm_gpio_fault_config_t (C++ struct), 1210
mcpwm_gpio_fault_config_t::active_level (C++ member), 1210
mcpwm_gpio_fault_config_t::flags (C++ member), 1210
mcpwm_gpio_fault_config_t::gpio_num (C++ member), 1210
mcpwm_gpio_fault_config_t::group_id (C++ member), 1210
mcpwm_gpio_fault_config_t::intr_priority (C++ member), 1210
mcpwm_gpio_fault_config_t::io_loop_back (C++ member), 1210
mcpwm_gpio_fault_config_t::pull_down (C++ member), 1210
mcpwm_gpio_fault_config_t::pull_up (C++ member), 1210
mcpwm_gpio_sync_src_config_t (C++ struct), 1212
mcpwm_gpio_sync_src_config_t::active_neg (C++ member), 1213
mcpwm_gpio_sync_src_config_t::flags (C++ member), 1213
mcpwm_gpio_sync_src_config_t::gpio_num (C++ member), 1213
mcpwm_gpio_sync_src_config_t::group_id (C++ member), 1213
mcpwm_gpio_sync_src_config_t::io_loop_back (C++ member), 1213
mcpwm_gpio_sync_src_config_t::pull_down (C++ member), 1213
mcpwm_gpio_sync_src_config_t::pull_up (C++ member), 1213
mcpwm_new_capture_channel (C++ function), 1215
mcpwm_new_capture_timer (C++ function), 1213
mcpwm_new_comparator (C++ function), 1200
mcpwm_new_generator (C++ function), 1202
mcpwm_new_gpio_fault (C++ function), 1209
mcpwm_new_gpio_sync_src (C++ function), 1211
mcpwm_new_operator (C++ function), 1197
mcpwm_new_soft_fault (C++ function), 1209
mcpwm_new_soft_sync_src (C++ function), 1212
mcpwm_new_timer (C++ function), 1193
mcpwm_new_timer_sync_src (C++ function), 1211
mcpwm_oper_handle_t (C++ type), 1219
mcpwm_operator_apply_carrier (C++ function), 1198
mcpwm_operator_brake_mode_t (C++ enum), 1222
mcpwm_operator_brake_mode_t::MCPWM_OPER_BRAKE_MOD (C++ enumerator), 1222
mcpwm_operator_brake_mode_t::MCPWM_OPER_BRAKE_MOD (C++ enumerator), 1223
mcpwm_operator_brake_mode_t::MCPWM_OPER_BRAKE_MOD (C++ enumerator), 1223
mcpwm_operator_config_t (C++ struct), 1198
mcpwm_operator_config_t::flags (C++ member), 1199
mcpwm_operator_config_t::group_id (C++ member), 1198
mcpwm_operator_config_t::intr_priority (C++ member), 1198
mcpwm_operator_config_t::update_dead_time_on_sync (C++ member), 1199
mcpwm_operator_config_t::update_dead_time_on_tep (C++ member), 1199
mcpwm_operator_config_t::update_dead_time_on_tez (C++ member), 1199
mcpwm_operator_config_t::update_gen_action_on_syn (C++ member), 1199
mcpwm_operator_config_t::update_gen_action_on_tep (C++ member), 1198
mcpwm_operator_config_t::update_gen_action_on_tez (C++ member), 1198
mcpwm_operator_connect_timer (C++ function), 1197
mcpwm_operator_event_callbacks_t (C++ struct), 1199
mcpwm_operator_event_callbacks_t::on_brake_cbc (C++ member), 1199
mcpwm_operator_event_callbacks_t::on_brake_ost (C++ member), 1199
mcpwm_operator_recover_from_fault (C++ function), 1197
mcpwm_operator_register_event_callbacks (C++ function), 1198
mcpwm_operator_set_brake_on_fault (C++ function), 1197
mcpwm_soft_fault_activate (C++ function), 1209

mcpwm_soft_fault_config_t (C++ struct), 1210
mcpwm_soft_sync_activate (C++ function), 1212
mcpwm_soft_sync_config_t (C++ struct), 1213
mcpwm_sync_handle_t (C++ type), 1220
mcpwm_timer_clock_source_t (C++ type), 1221
mcpwm_timer_config_t (C++ struct), 1195
mcpwm_timer_config_t::clk_src (C++ member), 1196
mcpwm_timer_config_t::count_mode (C++ member), 1196
mcpwm_timer_config_t::flags (C++ member), 1196
mcpwm_timer_config_t::group_id (C++ member), 1195
mcpwm_timer_config_t::intr_priority (C++ member), 1196
mcpwm_timer_config_t::period_ticks (C++ member), 1196
mcpwm_timer_config_t::resolution_hz (C++ member), 1196
mcpwm_timer_config_t::update_period_on_empty (C++ member), 1196
mcpwm_timer_config_t::update_period_on_sync (C++ member), 1196
mcpwm_timer_count_mode_t (C++ enum), 1221
mcpwm_timer_count_mode_t::MCPWM_TIMER_COUNT_MODE_DOWN (C++ enumerator), 1222
mcpwm_timer_count_mode_t::MCPWM_TIMER_COUNT_MODE_PAUSE (C++ enumerator), 1221
mcpwm_timer_count_mode_t::MCPWM_TIMER_COUNT_MODE_UP (C++ enumerator), 1221
mcpwm_timer_count_mode_t::MCPWM_TIMER_COUNT_MODE_UP_DOWN (C++ enumerator), 1222
mcpwm_timer_direction_t (C++ enum), 1221
mcpwm_timer_direction_t::MCPWM_TIMER_DIRECTION_DOWN (C++ enumerator), 1221
mcpwm_timer_direction_t::MCPWM_TIMER_DIRECTION_UP (C++ enumerator), 1221
mcpwm_timer_disable (C++ function), 1194
mcpwm_timer_enable (C++ function), 1194
mcpwm_timer_event_callbacks_t (C++ struct), 1195
mcpwm_timer_event_callbacks_t::on_empty (C++ member), 1195
mcpwm_timer_event_callbacks_t::on_full (C++ member), 1195
mcpwm_timer_event_callbacks_t::on_stop (C++ member), 1195
mcpwm_timer_event_cb_t (C++ type), 1220
mcpwm_timer_event_data_t (C++ struct), 1218
mcpwm_timer_event_data_t::count_value (C++ member), 1219
mcpwm_timer_event_data_t::direction (C++ member), 1219
mcpwm_timer_event_t (C++ enum), 1221
mcpwm_timer_event_t::MCPWM_TIMER_EVENT_EMPTY (C++ enumerator), 1221
mcpwm_timer_event_t::MCPWM_TIMER_EVENT_FULL (C++ enumerator), 1221
mcpwm_timer_event_t::MCPWM_TIMER_EVENT_INVALID (C++ enumerator), 1221
mcpwm_timer_handle_t (C++ type), 1219
mcpwm_timer_register_event_callbacks (C++ function), 1194
mcpwm_timer_set_period (C++ function), 1193
mcpwm_timer_set_phase_on_sync (C++ function), 1195
mcpwm_timer_start_stop (C++ function), 1194
mcpwm_timer_start_stop_cmd_t (C++ enum), 1222
mcpwm_timer_start_stop_cmd_t::MCPWM_TIMER_START_N (C++ enumerator), 1222
mcpwm_timer_start_stop_cmd_t::MCPWM_TIMER_START_S (C++ enumerator), 1222
mcpwm_timer_start_stop_cmd_t::MCPWM_TIMER_START_S (C++ enumerator), 1222
mcpwm_timer_start_stop_cmd_t::MCPWM_TIMER_STOP_EM (C++ enumerator), 1222
mcpwm_timer_start_stop_cmd_t::MCPWM_TIMER_STOP_FU (C++ enumerator), 1222
mcpwm_timer_sync_phase_config_t (C++ struct), 1196
mcpwm_timer_sync_phase_config_t::count_value (C++ member), 1196
mcpwm_timer_sync_phase_config_t::direction (C++ member), 1196
mcpwm_timer_sync_phase_config_t::sync_src (C++ member), 1196
mcpwm_timer_sync_src_config_t (C++ struct), 1212
mcpwm_timer_sync_src_config_t::flags (C++ member), 1212
mcpwm_timer_sync_src_config_t::propagate_input_sy (C++ member), 1212
mcpwm_timer_sync_src_config_t::timer_event (C++ member), 1212
mesh_addr_t (C++ union), 854
mesh_addr_t::addr (C++ member), 854
mesh_addr_t::mip (C++ member), 854
mesh_ap_cfg_t (C++ struct), 860
mesh_ap_cfg_t::max_connection (C++ member), 860
mesh_ap_cfg_t::nonmesh_max_connection (C++ member), 860
mesh_ap_cfg_t::password (C++ member), 860
MESH_ASSOC_FLAG_NETWORK_FREE (C macro), 864
MESH_ASSOC_FLAG_ROOT_FIXED (C macro), 864
MESH_ASSOC_FLAG_ROOTS_FOUND (C macro), 864
MESH_ASSOC_FLAG_VOTE_IN_PROGRESS (C macro), 864
mesh_cfg_t (C++ struct), 860

mesh_cfg_t::allow_channel_switch (*C++ member*), 860
mesh_cfg_t::channel (*C++ member*), 860
mesh_cfg_t::crypto_funcs (*C++ member*), 861
mesh_cfg_t::mesh_ap (*C++ member*), 861
mesh_cfg_t::mesh_id (*C++ member*), 860
mesh_cfg_t::router (*C++ member*), 861
MESH_DATA_DROP (*C macro*), 864
MESH_DATA_ENC (*C macro*), 864
MESH_DATA_FROMDS (*C macro*), 864
MESH_DATA_GROUP (*C macro*), 864
MESH_DATA_NONBLOCK (*C macro*), 864
MESH_DATA_P2P (*C macro*), 864
mesh_data_t (*C++ struct*), 859
mesh_data_t::data (*C++ member*), 859
mesh_data_t::proto (*C++ member*), 859
mesh_data_t::size (*C++ member*), 859
mesh_data_t::tos (*C++ member*), 859
MESH_DATA_TODS (*C macro*), 864
mesh_disconnect_reason_t (*C++ enum*), 868
mesh_disconnect_reason_t::MESH_REASON_CYCLIC (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_DIFF_ID (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_EMPTY_PASSWORD (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_IE_UNKNOWN (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_LEAF (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_PARENT_IDLE (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_PARENT_STOPPED (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_UNENCRYPTED (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_WORSE (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_ROOTS (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_SCAN_FAIL (*C++ enumerator*), 869
mesh_disconnect_reason_t::MESH_REASON_WAIVE_ROOT (*C++ enumerator*), 869
mesh_event_channel_switch_t (*C++ struct*), 856
mesh_event_channel_switch_t::channel (*C++ member*), 856
mesh_event_child_connected_t (*C++ type*), 865
mesh_event_child_disconnected_t (*C++ type*), 865
mesh_event_connected_t (*C++ struct*), 856
mesh_event_connected_t::connected (*C++ member*), 856
mesh_event_connected_t::duty (*C++ member*), 856
mesh_event_connected_t::self_layer (*C++ member*), 856
mesh_event_disconnected_t (*C++ type*), 865
mesh_event_find_network_t (*C++ struct*), 857
mesh_event_find_network_t::channel (*C++ member*), 857
mesh_event_find_network_t::router_bssid (*C++ member*), 857
mesh_event_id_t (*C++ enum*), 865
mesh_event_id_t::MESH_EVENT_CHANNEL_SWITCH (*C++ enumerator*), 865
mesh_event_id_t::MESH_EVENT_CHILD_CONNECTED (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_CHILD_DISCONNECTED (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_FIND_NETWORK (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_LAYER_CHANGE (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_MAX (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_NETWORK_STATE (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_NO_PARENT_FOUND (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_PARENT_CONNECTED (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_PARENT_DISCONNECTED (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_PS_CHILD_DUTY (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_PS_DEVICE_DUTY (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_PS_PARENT_DUTY (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_ROOT_ADDRESS (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_ROOT_ASKED_YIELD (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_ROOT_FIXED (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_ROOT_SWITCH_ACK (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_ROOT_SWITCH_REQ (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_ROUTER_SWITCH (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_ROUTING_TABLE_ADD (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_ROUTING_TABLE_REMOVE (*C++ enumerator*), 866
mesh_event_id_t::MESH_EVENT_SCAN_DONE (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_STARTED (*C++ enumerator*), 865
mesh_event_id_t::MESH_EVENT_STOP_RECONNECTION (*C++ enumerator*), 867
mesh_event_id_t::MESH_EVENT_STOPPED

(C++ *enumerator*), 865
mesh_event_id_t::MESH_EVENT_TODS_STATE (C++ *enumerator*), 866
mesh_event_id_t::MESH_EVENT_VOTE_STARTED (C++ *enumerator*), 866
mesh_event_id_t::MESH_EVENT_VOTE_STOPPED (C++ *enumerator*), 866
mesh_event_info_t (C++ *union*), 854
mesh_event_info_t::channel_switch (C++ *member*), 854
mesh_event_info_t::child_connected (C++ *member*), 854
mesh_event_info_t::child_disconnected (C++ *member*), 854
mesh_event_info_t::connected (C++ *member*), 854
mesh_event_info_t::disconnected (C++ *member*), 854
mesh_event_info_t::find_network (C++ *member*), 855
mesh_event_info_t::layer_change (C++ *member*), 855
mesh_event_info_t::network_state (C++ *member*), 855
mesh_event_info_t::no_parent (C++ *member*), 854
mesh_event_info_t::ps_duty (C++ *member*), 855
mesh_event_info_t::root_addr (C++ *member*), 855
mesh_event_info_t::root_conflict (C++ *member*), 855
mesh_event_info_t::root_fixed (C++ *member*), 855
mesh_event_info_t::router_switch (C++ *member*), 855
mesh_event_info_t::routing_table (C++ *member*), 854
mesh_event_info_t::scan_done (C++ *member*), 855
mesh_event_info_t::switch_req (C++ *member*), 855
mesh_event_info_t::toDS_state (C++ *member*), 855
mesh_event_info_t::vote_started (C++ *member*), 855
mesh_event_layer_change_t (C++ *struct*), 856
mesh_event_layer_change_t::new_layer (C++ *member*), 857
mesh_event_network_state_t (C++ *struct*), 858
mesh_event_network_state_t::is_rootless (C++ *member*), 858
mesh_event_no_parent_found_t (C++ *struct*), 856
mesh_event_no_parent_found_t::scan_times (C++ *member*), 856
mesh_event_ps_duty_t (C++ *struct*), 858
mesh_event_ps_duty_t::child_connected (C++ *member*), 859
mesh_event_ps_duty_t::duty (C++ *member*), 859
mesh_event_root_address_t (C++ *type*), 865
mesh_event_root_conflict_t (C++ *struct*), 857
mesh_event_root_conflict_t::addr (C++ *member*), 858
mesh_event_root_conflict_t::capacity (C++ *member*), 858
mesh_event_root_conflict_t::rssi (C++ *member*), 858
mesh_event_root_fixed_t (C++ *struct*), 858
mesh_event_root_fixed_t::is_fixed (C++ *member*), 858
mesh_event_root_switch_req_t (C++ *struct*), 857
mesh_event_root_switch_req_t::rc_addr (C++ *member*), 857
mesh_event_root_switch_req_t::reason (C++ *member*), 857
mesh_event_router_switch_t (C++ *type*), 865
mesh_event_routing_table_change_t (C++ *struct*), 858
mesh_event_routing_table_change_t::rt_size_change (C++ *member*), 858
mesh_event_routing_table_change_t::rt_size_new (C++ *member*), 858
mesh_event_scan_done_t (C++ *struct*), 858
mesh_event_scan_done_t::number (C++ *member*), 858
mesh_event_toDS_state_t (C++ *enum*), 870
mesh_event_toDS_state_t::MESH_TODS_REACHABLE (C++ *enumerator*), 870
mesh_event_toDS_state_t::MESH_TODS_UNREACHABLE (C++ *enumerator*), 870
mesh_event_vote_started_t (C++ *struct*), 857
mesh_event_vote_started_t::attempts (C++ *member*), 857
mesh_event_vote_started_t::rc_addr (C++ *member*), 857
mesh_event_vote_started_t::reason (C++ *member*), 857
MESH_INIT_CONFIG_DEFAULT (*C macro*), 865
MESH_MPS (*C macro*), 862
MESH_MTU (*C macro*), 862
MESH_OPT_RECV_DS_ADDR (*C macro*), 864
MESH_OPT_SEND_GROUP (*C macro*), 864
mesh_opt_t (C++ *struct*), 859
mesh_opt_t::len (C++ *member*), 859
mesh_opt_t::type (C++ *member*), 859
mesh_opt_t::val (C++ *member*), 859
mesh_proto_t (C++ *enum*), 868
mesh_proto_t::MESH_PROTO_AP (C++ *enumerator*), 868
mesh_proto_t::MESH_PROTO_BIN (C++ *enumerator*), 868

mesh_proto_t::MESH_PROTO_HTTP (C++ enumerator), 868
mesh_proto_t::MESH_PROTO_JSON (C++ enumerator), 868
mesh_proto_t::MESH_PROTO_MQTT (C++ enumerator), 868
mesh_proto_t::MESH_PROTO_STA (C++ enumerator), 868
MESH_PS_DEVICE_DUTY_DEMAND (C macro), 865
MESH_PS_DEVICE_DUTY_REQUEST (C macro), 865
MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE (C macro), 865
MESH_PS_NETWORK_DUTY_APPLIED_UPLINK (C macro), 865
MESH_PS_NETWORK_DUTY_MASTER (C macro), 865
mesh_rc_config_t (C++ union), 855
mesh_rc_config_t::attempts (C++ member), 855
mesh_rc_config_t::rc_addr (C++ member), 855
MESH_ROOT_LAYER (C macro), 862
mesh_router_t (C++ struct), 859
mesh_router_t::allow_router_switch (C++ member), 860
mesh_router_t::bssid (C++ member), 860
mesh_router_t::password (C++ member), 860
mesh_router_t::ssid (C++ member), 860
mesh_router_t::ssid_len (C++ member), 860
mesh_rx_pending_t (C++ struct), 862
mesh_rx_pending_t::toDS (C++ member), 862
mesh_rx_pending_t::toSelf (C++ member), 862
mesh_tos_t (C++ enum), 868
mesh_tos_t::MESH_TOS_DEF (C++ enumerator), 868
mesh_tos_t::MESH_TOS_E2E (C++ enumerator), 868
mesh_tos_t::MESH_TOS_P2P (C++ enumerator), 868
mesh_tx_pending_t (C++ struct), 861
mesh_tx_pending_t::broadcast (C++ member), 861
mesh_tx_pending_t::mgmt (C++ member), 861
mesh_tx_pending_t::to_child (C++ member), 861
mesh_tx_pending_t::to_child_p2p (C++ member), 861
mesh_tx_pending_t::to_parent (C++ member), 861
mesh_tx_pending_t::to_parent_p2p (C++ member), 861
mesh_type_t (C++ enum), 867
mesh_type_t::MESH_IDLE (C++ enumerator), 867
mesh_type_t::MESH_LEAF (C++ enumerator), 867
mesh_type_t::MESH_NODE (C++ enumerator), 867
mesh_type_t::MESH_ROOT (C++ enumerator), 867
mesh_type_t::MESH_STA (C++ enumerator), 867
mesh_vote_reason_t (C++ enum), 868
mesh_vote_reason_t::MESH_VOTE_REASON_CHILD_INITIA (C++ enumerator), 868
mesh_vote_reason_t::MESH_VOTE_REASON_ROOT_INITIAT (C++ enumerator), 868
mesh_vote_t (C++ struct), 861
mesh_vote_t::config (C++ member), 861
mesh_vote_t::is_rc_specified (C++ member), 861
mesh_vote_t::percentage (C++ member), 861
MessageBufferHandle_t (C++ type), 2011
mip_t (C++ struct), 856
mip_t::ip4 (C++ member), 856
mip_t::port (C++ member), 856
MQTT_ERROR_TYPE_ESP_TLS (C macro), 137
mqtt_event_callback_t (C++ type), 138
multi_heap_aligned_alloc (C++ function), 2040
multi_heap_aligned_free (C++ function), 2041
multi_heap_check (C++ function), 2042
multi_heap_dump (C++ function), 2042
multi_heap_free (C++ function), 2041
multi_heap_free_size (C++ function), 2042
multi_heap_get_allocated_size (C++ function), 2041
multi_heap_get_info (C++ function), 2043
multi_heap_handle_t (C++ type), 2043
multi_heap_info_t (C++ struct), 2043
multi_heap_info_t::allocated_blocks (C++ member), 2043
multi_heap_info_t::free_blocks (C++ member), 2043
multi_heap_info_t::largest_free_block (C++ member), 2043
multi_heap_info_t::minimum_free_bytes (C++ member), 2043
multi_heap_info_t::total_allocated_bytes (C++ member), 2043
multi_heap_info_t::total_blocks (C++ member), 2043
multi_heap_info_t::total_free_bytes (C++ member), 2043
multi_heap_malloc (C++ function), 2041
multi_heap_minimum_free_size (C++ function), 2042
multi_heap_realloc (C++ function), 2041
multi_heap_register (C++ function), 2041
multi_heap_set_lock (C++ function), 2042

N

name_uuid (C++ struct), 1688
name_uuid::name (C++ member), 1688

name_uuid::uuid (*C++ member*), 1688
neighbor_rep_request_cb (*C++ type*), 938
non_pref_chan (*C++ struct*), 940
non_pref_chan::chan (*C++ member*), 940
non_pref_chan::oper_class (*C++ member*), 940
non_pref_chan::preference (*C++ member*), 940
non_pref_chan::reason (*C++ member*), 940
non_pref_chan_reason (*C++ enum*), 940
non_pref_chan_reason::NON_PREF_CHAN_REASON_EXTERNAL (*C++ enumerator*), 941
non_pref_chan_reason::NON_PREF_CHAN_REASON_INTERNAL (*C++ enumerator*), 941
non_pref_chan_reason::NON_PREF_CHAN_REASON_NONE (*C++ enumerator*), 941
non_pref_chan_reason::NON_PREF_CHAN_REASON_UNSPECIFIED (*C++ enumerator*), 940
non_pref_chan_s (*C++ struct*), 940
non_pref_chan_s::chan (*C++ member*), 940
non_pref_chan_s::non_pref_chan_num (*C++ member*), 940
nvs_close (*C++ function*), 1742
nvs_commit (*C++ function*), 1742
NVS_DEFAULT_PART_NAME (*C macro*), 1747
nvs_entry_find (*C++ function*), 1743
nvs_entry_info (*C++ function*), 1744
nvs_entry_info_t (*C++ struct*), 1745
nvs_entry_info_t::key (*C++ member*), 1745
nvs_entry_info_t::namespace_name (*C++ member*), 1745
nvs_entry_info_t::type (*C++ member*), 1745
nvs_entry_next (*C++ function*), 1744
nvs_erase_all (*C++ function*), 1742
nvs_erase_key (*C++ function*), 1741
nvs_flash_deinit (*C++ function*), 1734
nvs_flash_deinit_partition (*C++ function*), 1734
nvs_flash_erase (*C++ function*), 1734
nvs_flash_erase_partition (*C++ function*), 1735
nvs_flash_erase_partition_ptr (*C++ function*), 1735
nvs_flash_generate_keys (*C++ function*), 1736
nvs_flash_init (*C++ function*), 1733
nvs_flash_init_partition (*C++ function*), 1734
nvs_flash_init_partition_ptr (*C++ function*), 1734
nvs_flash_read_security_cfg (*C++ function*), 1736
nvs_flash_secure_init (*C++ function*), 1735
nvs_flash_secure_init_partition (*C++ function*), 1735
nvs_get_blob (*C++ function*), 1740
nvs_get_i16 (*C++ function*), 1739
nvs_get_i32 (*C++ function*), 1739
nvs_get_i64 (*C++ function*), 1739
nvs_get_i8 (*C++ function*), 1738
nvs_get_stats (*C++ function*), 1742
nvs_get_str (*C++ function*), 1739
nvs_get_u16 (*C++ function*), 1739
nvs_get_u32 (*C++ function*), 1739
nvs_get_u64 (*C++ function*), 1739
nvs_get_u8 (*C++ function*), 1738
nvs_get_used_entry_count (*C++ function*), 1743
nvs_handle (*C++ type*), 1747
nvs_handle_t (*C++ type*), 1747
NVS_INVALID_HANDLE (*C++ type*), 1747
NVS_KEY_NAME_MAX_SIZE (*C macro*), 1747
NVS_KEY_SIZE (*C macro*), 1736
nvs_open (*C++ function*), 1740
nvs_open_from_partition (*C++ function*), 1740
nvs_open_mode (*C++ type*), 1747
nvs_open_mode_t (*C++ enum*), 1747
nvs_open_mode_t::NVS_READONLY (*C++ enumerator*), 1747
nvs_open_mode_t::NVS_READWRITE (*C++ enumerator*), 1747
NVS_PART_NAME_MAX_SIZE (*C macro*), 1747
nvs_release_iterator (*C++ function*), 1744
nvs_sec_cfg_t (*C++ struct*), 1736
nvs_sec_cfg_t::eky (*C++ member*), 1736
nvs_sec_cfg_t::tky (*C++ member*), 1736
nvs_set_blob (*C++ function*), 1741
nvs_set_i16 (*C++ function*), 1737
nvs_set_i32 (*C++ function*), 1737
nvs_set_i64 (*C++ function*), 1737
nvs_set_i8 (*C++ function*), 1737
nvs_set_str (*C++ function*), 1737
nvs_set_u16 (*C++ function*), 1737
nvs_set_u32 (*C++ function*), 1737
nvs_set_u64 (*C++ function*), 1737
nvs_set_u8 (*C++ function*), 1737
nvs_stats_t (*C++ struct*), 1745
nvs_stats_t::free_entries (*C++ member*), 1745
nvs_stats_t::namespace_count (*C++ member*), 1745
nvs_stats_t::total_entries (*C++ member*), 1745
nvs_stats_t::used_entries (*C++ member*), 1745
nvs_type_t (*C++ enum*), 1748
nvs_type_t::NVS_TYPE_ANY (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_BLOB (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_I16 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_I32 (*C++ enumerator*), 1748

nvs_type_t::NVS_TYPE_I64 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_I8 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_STR (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_U16 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_U32 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_U64 (*C++ enumerator*), 1748
nvs_type_t::NVS_TYPE_U8 (*C++ enumerator*), 1748

O

OPCODE_ADC (*C macro*), 2171
OPCODE_ALU (*C macro*), 2172
OPCODE_BRANCH (*C macro*), 2173
OPCODE_DELAY (*C macro*), 2171
OPCODE_END (*C macro*), 2173
OPCODE_HALT (*C macro*), 2174
OPCODE_I2C (*C macro*), 2171
OPCODE_LD (*C macro*), 2174
OPCODE_MACRO (*C macro*), 2174
OPCODE_RD_REG (*C macro*), 2171
OPCODE_ST (*C macro*), 2172
OPCODE_TSENS (*C macro*), 2174
OPCODE_WR_REG (*C macro*), 2171
OTA_SIZE_UNKNOWN (*C macro*), 2106
OTA_WITH_SEQUENTIAL_WRITES (*C macro*), 2106

P

pcnt_chan_config_t (*C++ struct*), 1235
pcnt_chan_config_t::edge_gpio_num (*C++ member*), 1235
pcnt_chan_config_t::flags (*C++ member*), 1236
pcnt_chan_config_t::invert_edge_input (*C++ member*), 1236
pcnt_chan_config_t::invert_level_input (*C++ member*), 1236
pcnt_chan_config_t::io_loop_back (*C++ member*), 1236
pcnt_chan_config_t::level_gpio_num (*C++ member*), 1236
pcnt_chan_config_t::virt_edge_io_level (*C++ member*), 1236
pcnt_chan_config_t::virt_level_io_level (*C++ member*), 1236
pcnt_channel_edge_action_t (*C++ enum*), 1237
pcnt_channel_edge_action_t::PCNT_CHANNEL_EDGE_ACTION_DECREASE (*C++ enumerator*), 1237
pcnt_channel_edge_action_t::PCNT_CHANNEL_EDGE_ACTION_HOLD (*C++ enumerator*), 1237
pcnt_channel_edge_action_t::PCNT_CHANNEL_EDGE_ACTION_INCREASE (*C++ enumerator*), 1237
pcnt_channel_handle_t (*C++ type*), 1236
pcnt_channel_level_action_t (*C++ enum*), 1237
pcnt_channel_level_action_t::PCNT_CHANNEL_LEVEL_A (*C++ enumerator*), 1237
pcnt_channel_level_action_t::PCNT_CHANNEL_LEVEL_A (*C++ enumerator*), 1237
pcnt_channel_level_action_t::PCNT_CHANNEL_LEVEL_A (*C++ enumerator*), 1237
pcnt_channel_set_edge_action (*C++ function*), 1234
pcnt_channel_set_level_action (*C++ function*), 1234
pcnt_del_channel (*C++ function*), 1234
pcnt_del_unit (*C++ function*), 1229
pcnt_event_callbacks_t (*C++ struct*), 1235
pcnt_event_callbacks_t::on_reach (*C++ member*), 1235
pcnt_glitch_filter_config_t (*C++ struct*), 1236
pcnt_glitch_filter_config_t::max_glitch_ns (*C++ member*), 1236
pcnt_new_channel (*C++ function*), 1233
pcnt_new_unit (*C++ function*), 1229
pcnt_unit_add_watch_point (*C++ function*), 1233
pcnt_unit_clear_count (*C++ function*), 1232
pcnt_unit_config_t (*C++ struct*), 1235
pcnt_unit_config_t::accum_count (*C++ member*), 1235
pcnt_unit_config_t::flags (*C++ member*), 1235
pcnt_unit_config_t::high_limit (*C++ member*), 1235
pcnt_unit_config_t::intr_priority (*C++ member*), 1235
pcnt_unit_config_t::low_limit (*C++ member*), 1235
pcnt_unit_disable (*C++ function*), 1230
pcnt_unit_enable (*C++ function*), 1230
pcnt_unit_get_count (*C++ function*), 1232
pcnt_unit_handle_t (*C++ type*), 1236
pcnt_unit_register_event_callbacks (*C++ function*), 1232
pcnt_unit_remove_watch_point (*C++ function*), 1233
pcnt_unit_set_glitch_filter (*C++ function*), 1229
pcnt_unit_start (*C++ function*), 1231
pcnt_unit_stop (*C++ function*), 1231
pcnt_unit_zero_cross_mode_t (*C++ enum*), 1237
pcnt_unit_zero_cross_mode_t::PCNT_UNIT_ZERO_CROSS (*C++ enumerator*), 1237
pcnt_unit_zero_cross_mode_t::PCNT_UNIT_ZERO_CROSS (*C++ enumerator*), 1237
pcnt_unit_zero_cross_mode_t::PCNT_UNIT_ZERO_CROSS (*C++ enumerator*), 1237

pcnt_unit_zero_cross_mode_t::PCNT_UNIT_ZERO_CROSS_NEG_ZERO (C++ enumerator), 1237
pcnt_unit_zero_cross_mode_t::PCNT_UNIT_ZERO_CROSS_POS_ZERO (C++ enumerator), 1237
pcnt_watch_cb_t (C++ type), 1236
pcnt_watch_event_data_t (C++ struct), 1234
pcnt_watch_event_data_t::watch_point_value (C++ member), 1234
pcnt_watch_event_data_t::zero_cross_mode (C++ member), 1235
pcQueueGetName (C++ function), 1942
pcTaskGetName (C++ function), 1917
pcTimerGetName (C++ function), 1974
PendedFunction_t (C++ type), 1986
phy_802_3_t (C++ struct), 979
phy_802_3_t::addr (C++ member), 979
phy_802_3_t::autonego_timeout_ms (C++ member), 979
phy_802_3_t::eth (C++ member), 979
phy_802_3_t::link_status (C++ member), 979
phy_802_3_t::parent (C++ member), 979
phy_802_3_t::reset_gpio_num (C++ member), 980
phy_802_3_t::reset_timeout_ms (C++ member), 979
phy_init_param_set (C++ function), 2439
phy_wifi_enable_set (C++ function), 2439
PIN_LEN (C macro), 937
protocomm_add_endpoint (C++ function), 1681
protocomm_ble_config (C++ struct), 1688
protocomm_ble_config::ble_bonding (C++ member), 1689
protocomm_ble_config::ble_link_encryption (C++ member), 1689
protocomm_ble_config::ble_sm_sc (C++ member), 1689
protocomm_ble_config::device_name (C++ member), 1688
protocomm_ble_config::manufacturer_data (C++ member), 1689
protocomm_ble_config::manufacturer_data_len (C++ member), 1689
protocomm_ble_config::nu_lookup (C++ member), 1689
protocomm_ble_config::nu_lookup_count (C++ member), 1689
protocomm_ble_config::service_uuid (C++ member), 1689
protocomm_ble_config_t (C++ type), 1689
protocomm_ble_name_uuid_t (C++ type), 1689
protocomm_ble_start (C++ function), 1688
protocomm_ble_stop (C++ function), 1688
protocomm_close_session (C++ function), 1682
protocomm_delete (C++ function), 1680
protocomm_http_server_config_t (C++ struct), 1687
protocomm_http_server_config_t::port (C++ member), 1687
protocomm_http_server_config_t::stack_size (C++ member), 1687
protocomm_http_server_config_t::task_priority (C++ member), 1687
protocomm_httpd_config_data_t (C++ union), 1687
protocomm_httpd_config_data_t::config (C++ member), 1687
protocomm_httpd_config_data_t::handle (C++ member), 1687
protocomm_httpd_config_t (C++ struct), 1687
protocomm_httpd_config_t::data (C++ member), 1687
protocomm_httpd_config_t::ext_handle_provided (C++ member), 1687
PROTOCOMM_HTTPD_DEFAULT_CONFIG (C macro), 1687
protocomm_httpd_start (C++ function), 1686
protocomm_httpd_stop (C++ function), 1686
protocomm_new (C++ function), 1680
protocomm_open_session (C++ function), 1681
protocomm_remove_endpoint (C++ function), 1681
protocomm_req_handle (C++ function), 1682
protocomm_req_handler_t (C++ type), 1684
protocomm_security (C++ struct), 1685
protocomm_security1_params (C++ struct), 1684
protocomm_security1_params::data (C++ member), 1684
protocomm_security1_params::len (C++ member), 1684
protocomm_security1_params_t (C++ type), 1685
protocomm_security2_params (C++ struct), 1684
protocomm_security2_params::salt (C++ member), 1684
protocomm_security2_params::salt_len (C++ member), 1684
protocomm_security2_params::verifier (C++ member), 1684
protocomm_security2_params::verifier_len (C++ member), 1685
protocomm_security2_params_t (C++ type), 1685
protocomm_security::cleanup (C++ member), 1685
protocomm_security::close_transport_session (C++ member), 1685
protocomm_security::decrypt (C++ member), 1685
protocomm_security::encrypt (C++ member), 1685
protocomm_security::init (C++ member), 1685
protocomm_security::new_transport_session (C++ member), 1685

protocomm_security::security_req_handler (C++ member), 1685
protocomm_security::ver (C++ member), 1685
protocomm_security_handle_t (C++ type), 1686
protocomm_security_pop_t (C++ type), 1685
protocomm_security_t (C++ type), 1686
protocomm_set_security (C++ function), 1682
protocomm_set_version (C++ function), 1683
protocomm_t (C++ type), 1684
protocomm_unset_security (C++ function), 1683
protocomm_unset_version (C++ function), 1683
PROV_DATA_FLAGS_FLAG (C macro), 579
PROV_DATA_IV_INDEX_FLAG (C macro), 579
PROV_DATA_NET_IDX_FLAG (C macro), 579
psk_hint_key_t (C++ type), 151
psk_key_hint (C++ struct), 148
psk_key_hint::hint (C++ member), 148
psk_key_hint::key (C++ member), 148
psk_key_hint::key_size (C++ member), 148
PTHREAD_STACK_MIN (C macro), 2120
pvTaskGetThreadLocalStoragePointer (C++ function), 1918
pvTimerGetTimerID (C++ function), 1971
pxTaskGetStackStart (C++ function), 1918

Q

QueueHandle_t (C++ type), 1954
QueueSetHandle_t (C++ type), 1954
QueueSetMemberHandle_t (C++ type), 1954

R

R0 (C macro), 2171
R1 (C macro), 2171
R2 (C macro), 2171
R3 (C macro), 2171
RD_REG_PERIPH_RTC_CNTL (C macro), 2171
RD_REG_PERIPH_RTC_I2C (C macro), 2171
RD_REG_PERIPH_RTC_IO (C macro), 2171
RD_REG_PERIPH_SENS (C macro), 2171
RingbufferType_t (C++ enum), 2028
RingbufferType_t::RINGBUF_TYPE_ALLOWSPLIT (C++ enumerator), 2028
RingbufferType_t::RINGBUF_TYPE_BYTEBUF (C++ enumerator), 2028
RingbufferType_t::RINGBUF_TYPE_MAX (C++ enumerator), 2029
RingbufferType_t::RINGBUF_TYPE_NOSPLIT (C++ enumerator), 2028
RingbufHandle_t (C++ type), 2028
rmt_alloc_encoder_mem (C++ function), 1261
rmt_apply_carrier (C++ function), 1258
rmt_bytes_encoder_config_t (C++ struct), 1262
rmt_bytes_encoder_config_t::bit0 (C++ member), 1262
rmt_bytes_encoder_config_t::bit1 (C++ member), 1262
rmt_bytes_encoder_config_t::flags (C++ member), 1262
rmt_bytes_encoder_config_t::msb_first (C++ member), 1262
rmt_bytes_encoder_update_config (C++ function), 1260
rmt_carrier_config_t (C++ struct), 1259
rmt_carrier_config_t::always_on (C++ member), 1260
rmt_carrier_config_t::duty_cycle (C++ member), 1259
rmt_carrier_config_t::flags (C++ member), 1260
rmt_carrier_config_t::frequency_hz (C++ member), 1259
rmt_carrier_config_t::polarity_active_low (C++ member), 1259
rmt_channel_handle_t (C++ type), 1263
rmt_clock_source_t (C++ type), 1264
rmt_copy_encoder_config_t (C++ struct), 1262
rmt_del_channel (C++ function), 1258
rmt_del_encoder (C++ function), 1261
rmt_del_sync_manager (C++ function), 1253
rmt_disable (C++ function), 1259
rmt_enable (C++ function), 1259
rmt_encode_state_t (C++ enum), 1262
rmt_encode_state_t::RMT_ENCODING_COMPLETE (C++ enumerator), 1262
rmt_encode_state_t::RMT_ENCODING_MEM_FULL (C++ enumerator), 1263
rmt_encode_state_t::RMT_ENCODING_RESET (C++ enumerator), 1262
rmt_encoder_handle_t (C++ type), 1263
rmt_encoder_reset (C++ function), 1261
rmt_encoder_t (C++ struct), 1261
rmt_encoder_t (C++ type), 1262
rmt_encoder_t::del (C++ member), 1262
rmt_encoder_t::encode (C++ member), 1261
rmt_encoder_t::reset (C++ member), 1262
rmt_new_bytes_encoder (C++ function), 1260
rmt_new_copy_encoder (C++ function), 1260
rmt_new_rx_channel (C++ function), 1256
rmt_new_sync_manager (C++ function), 1253
rmt_new_tx_channel (C++ function), 1252
rmt_receive (C++ function), 1256
rmt_receive_config_t (C++ struct), 1258
rmt_receive_config_t::signal_range_max_ns (C++ member), 1258
rmt_receive_config_t::signal_range_min_ns (C++ member), 1258
rmt_rx_channel_config_t (C++ struct), 1257
rmt_rx_channel_config_t::clk_src (C++ member), 1257

rmt_rx_channel_config_t::flags (C++ member), 1258
rmt_rx_channel_config_t::gpio_num (C++ member), 1257
rmt_rx_channel_config_t::intr_priority (C++ member), 1258
rmt_rx_channel_config_t::invert_in (C++ member), 1258
rmt_rx_channel_config_t::io_loop_back (C++ member), 1258
rmt_rx_channel_config_t::mem_block_symbols (C++ member), 1258
rmt_rx_channel_config_t::resolution_hz (C++ member), 1257
rmt_rx_channel_config_t::with_dma (C++ member), 1258
rmt_rx_done_callback_t (C++ type), 1264
rmt_rx_done_event_data_t (C++ struct), 1263
rmt_rx_done_event_data_t::num_symbols (C++ member), 1263
rmt_rx_done_event_data_t::received_symbols (C++ member), 1263
rmt_rx_event_callbacks_t (C++ struct), 1257
rmt_rx_event_callbacks_t::on_recv_done (C++ member), 1257
rmt_rx_register_event_callbacks (C++ function), 1257
rmt_symbol_word_t (C++ union), 1264
rmt_symbol_word_t::duration0 (C++ member), 1264
rmt_symbol_word_t::duration1 (C++ member), 1264
rmt_symbol_word_t::level0 (C++ member), 1264
rmt_symbol_word_t::level1 (C++ member), 1264
rmt_symbol_word_t::val (C++ member), 1264
rmt_symbol_word_t::[anonymous] (C++ member), 1264
rmt_sync_manager_config_t (C++ struct), 1255
rmt_sync_manager_config_t::array_size (C++ member), 1256
rmt_sync_manager_config_t::tx_channel_array (C++ member), 1256
rmt_sync_manager_handle_t (C++ type), 1263
rmt_sync_reset (C++ function), 1254
rmt_transmit (C++ function), 1252
rmt_transmit_config_t (C++ struct), 1255
rmt_transmit_config_t::eot_level (C++ member), 1255
rmt_transmit_config_t::flags (C++ member), 1255
rmt_transmit_config_t::loop_count (C++ member), 1255
rmt_transmit_config_t::queue_nonblocking (C++ member), 1255
rmt_tx_channel_config_t (C++ struct), 1254
rmt_tx_channel_config_t::clk_src (C++ member), 1254
rmt_tx_channel_config_t::flags (C++ member), 1255
rmt_tx_channel_config_t::gpio_num (C++ member), 1254
rmt_tx_channel_config_t::intr_priority (C++ member), 1255
rmt_tx_channel_config_t::invert_out (C++ member), 1255
rmt_tx_channel_config_t::io_loop_back (C++ member), 1255
rmt_tx_channel_config_t::io_od_mode (C++ member), 1255
rmt_tx_channel_config_t::mem_block_symbols (C++ member), 1254
rmt_tx_channel_config_t::resolution_hz (C++ member), 1254
rmt_tx_channel_config_t::trans_queue_depth (C++ member), 1255
rmt_tx_channel_config_t::with_dma (C++ member), 1255
rmt_tx_done_callback_t (C++ type), 1263
rmt_tx_done_event_data_t (C++ struct), 1263
rmt_tx_done_event_data_t::num_symbols (C++ member), 1263
rmt_tx_event_callbacks_t (C++ struct), 1254
rmt_tx_event_callbacks_t::on_trans_done (C++ member), 1254
rmt_tx_register_event_callbacks (C++ function), 1253
rmt_tx_wait_all_done (C++ function), 1252
rtc_gpio_deinit (C++ function), 1068
rtc_gpio_force_hold_dis_all (C++ function), 1071
rtc_gpio_force_hold_en_all (C++ function), 1071
rtc_gpio_get_drive_capability (C++ function), 1070
rtc_gpio_get_level (C++ function), 1068
rtc_gpio_hold_dis (C++ function), 1070
rtc_gpio_hold_en (C++ function), 1070
rtc_gpio_init (C++ function), 1068
RTC_GPIO_IS_VALID_GPIO (C macro), 1071
rtc_gpio_is_valid_gpio (C++ function), 1068
rtc_gpio_isolate (C++ function), 1070
rtc_gpio_mode_t (C++ enum), 1071
rtc_gpio_mode_t::RTC_GPIO_MODE_DISABLED (C++ enumerator), 1071
rtc_gpio_mode_t::RTC_GPIO_MODE_INPUT_ONLY (C++ enumerator), 1071
rtc_gpio_mode_t::RTC_GPIO_MODE_INPUT_OUTPUT (C++ enumerator), 1071
rtc_gpio_mode_t::RTC_GPIO_MODE_INPUT_OUTPUT_OD (C++ enumerator), 1072
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_OD (C++ enumerator), 1072
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_ONLY

(C++ *enumerator*), 1071
rtc_gpio_pulldown_dis (C++ *function*), 1069
rtc_gpio_pulldown_en (C++ *function*), 1069
rtc_gpio_pullup_dis (C++ *function*), 1069
rtc_gpio_pullup_en (C++ *function*), 1069
rtc_gpio_set_direction (C++ *function*), 1069
rtc_gpio_set_direction_in_sleep (C++ *function*), 1069
rtc_gpio_set_drive_capability (C++ *function*), 1070
rtc_gpio_set_level (C++ *function*), 1068
rtc_gpio_wakeup_disable (C++ *function*), 1071
rtc_gpio_wakeup_enable (C++ *function*), 1071
rtc_io_number_get (C++ *function*), 1068
RTC_SLOW_MEM (C *macro*), 2184

S

sdio_event_cb_t (C++ *type*), 1289
sdio_slave_buf_handle_t (C++ *type*), 1289
sdio_slave_clear_host_int (C++ *function*), 1287
sdio_slave_config_t (C++ *struct*), 1288
sdio_slave_config_t::event_cb (C++ *member*), 1288
sdio_slave_config_t::flags (C++ *member*), 1288
sdio_slave_config_t::recv_buffer_size (C++ *member*), 1288
sdio_slave_config_t::send_queue_size (C++ *member*), 1288
sdio_slave_config_t::sending_mode (C++ *member*), 1288
sdio_slave_config_t::timing (C++ *member*), 1288
sdio_slave_deinit (C++ *function*), 1284
SDIO_SLAVE_FLAG_DAT2_DISABLED (C *macro*), 1288
SDIO_SLAVE_FLAG_DEFAULT_SPEED (C *macro*), 1289
SDIO_SLAVE_FLAG_HIGH_SPEED (C *macro*), 1289
SDIO_SLAVE_FLAG_HOST_INTR_DISABLED (C *macro*), 1288
SDIO_SLAVE_FLAG_INTERNAL_PULLUP (C *macro*), 1289
sdio_slave_get_host_intena (C++ *function*), 1287
sdio_slave_hostint_t (C++ *enum*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT0 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT1 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT2 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT3 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT4 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT5 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT6 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_BIT7 (C++ *enumerator*), 1283
sdio_slave_hostint_t::SDIO_SLAVE_HOSTINT_SEND_NEW (C++ *enumerator*), 1283
sdio_slave_initialize (C++ *function*), 1284
sdio_slave_read_reg (C++ *function*), 1287
sdio_slave_recv (C++ *function*), 1286
sdio_slave_recv_get_buf (C++ *function*), 1286
sdio_slave_recv_load_buf (C++ *function*), 1285
SDIO_SLAVE_RECV_MAX_BUFFER (C *macro*), 1288
sdio_slave_recv_packet (C++ *function*), 1285
sdio_slave_recv_register_buf (C++ *function*), 1285
sdio_slave_recv_unregister_buf (C++ *function*), 1285
sdio_slave_reset (C++ *function*), 1285
sdio_slave_send_get_finished (C++ *function*), 1286
sdio_slave_send_host_int (C++ *function*), 1287
sdio_slave_send_queue (C++ *function*), 1286
sdio_slave_sending_mode_t (C++ *enum*), 1284
sdio_slave_sending_mode_t::SDIO_SLAVE_SEND_PACKET (C++ *enumerator*), 1284
sdio_slave_sending_mode_t::SDIO_SLAVE_SEND_STREAM (C++ *enumerator*), 1284
sdio_slave_set_host_intena (C++ *function*), 1287
sdio_slave_start (C++ *function*), 1284
sdio_slave_stop (C++ *function*), 1285
sdio_slave_timing_t (C++ *enum*), 1283
sdio_slave_timing_t::SDIO_SLAVE_TIMING_NSEND_NSAM (C++ *enumerator*), 1284
sdio_slave_timing_t::SDIO_SLAVE_TIMING_NSEND_PSAM (C++ *enumerator*), 1284
sdio_slave_timing_t::SDIO_SLAVE_TIMING_PSEND_NSAM (C++ *enumerator*), 1284
sdio_slave_timing_t::SDIO_SLAVE_TIMING_PSEND_PSAM (C++ *enumerator*), 1283
sdio_slave_transmit (C++ *function*), 1287
sdio_slave_wait_int (C++ *function*), 1287
sdio_slave_write_reg (C++ *function*), 1287
sdm_channel_disable (C++ *function*), 1293
sdm_channel_enable (C++ *function*), 1292
sdm_channel_handle_t (C++ *type*), 1294
sdm_channel_set_duty (C++ *function*), 1293
sdm_clock_source_t (C++ *type*), 1294
sdm_config_t (C++ *struct*), 1293
sdm_config_t::clk_src (C++ *member*), 1294
sdm_config_t::flags (C++ *member*), 1294
sdm_config_t::gpio_num (C++ *member*), 1294

sdm_config_t::invert_out (C++ member), 1294
sdm_config_t::io_loop_back (C++ member), 1294
sdm_config_t::sample_rate_hz (C++ member), 1294
sdm_del_channel (C++ function), 1292
sdm_new_channel (C++ function), 1291
sdmmc_can_discard (C++ function), 1756
sdmmc_can_trim (C++ function), 1756
sdmmc_card_init (C++ function), 1755
sdmmc_card_print_info (C++ function), 1755
sdmmc_card_t (C++ struct), 1764
sdmmc_card_t::cid (C++ member), 1764
sdmmc_card_t::csd (C++ member), 1765
sdmmc_card_t::ext_csd (C++ member), 1765
sdmmc_card_t::host (C++ member), 1764
sdmmc_card_t::is_ddr (C++ member), 1765
sdmmc_card_t::is_mem (C++ member), 1765
sdmmc_card_t::is_mmc (C++ member), 1765
sdmmc_card_t::is_sdio (C++ member), 1765
sdmmc_card_t::log_bus_width (C++ member), 1765
sdmmc_card_t::max_freq_khz (C++ member), 1765
sdmmc_card_t::num_io_functions (C++ member), 1765
sdmmc_card_t::ocr (C++ member), 1764
sdmmc_card_t::raw_cid (C++ member), 1765
sdmmc_card_t::rca (C++ member), 1765
sdmmc_card_t::reserved (C++ member), 1765
sdmmc_card_t::scr (C++ member), 1765
sdmmc_card_t::ssr (C++ member), 1765
sdmmc_cid_t (C++ struct), 1760
sdmmc_cid_t::date (C++ member), 1761
sdmmc_cid_t::mfg_id (C++ member), 1761
sdmmc_cid_t::name (C++ member), 1761
sdmmc_cid_t::oem_id (C++ member), 1761
sdmmc_cid_t::revision (C++ member), 1761
sdmmc_cid_t::serial (C++ member), 1761
sdmmc_command_t (C++ struct), 1762
sdmmc_command_t::arg (C++ member), 1763
sdmmc_command_t::blklen (C++ member), 1763
sdmmc_command_t::data (C++ member), 1763
sdmmc_command_t::datalen (C++ member), 1763
sdmmc_command_t::error (C++ member), 1763
sdmmc_command_t::flags (C++ member), 1763
sdmmc_command_t::opcode (C++ member), 1763
sdmmc_command_t::response (C++ member), 1763
sdmmc_command_t::timeout_ms (C++ member), 1763
sdmmc_csd_t (C++ struct), 1760
sdmmc_csd_t::capacity (C++ member), 1760
sdmmc_csd_t::card_command_class (C++ member), 1760
sdmmc_csd_t::csd_ver (C++ member), 1760
sdmmc_csd_t::mmc_ver (C++ member), 1760
sdmmc_csd_t::read_block_len (C++ member), 1760
sdmmc_csd_t::sector_size (C++ member), 1760
sdmmc_csd_t::tr_speed (C++ member), 1760
sdmmc_erase_arg_t (C++ enum), 1766
sdmmc_erase_arg_t::SDMMC_DISCARD_ARG (C++ enumerator), 1767
sdmmc_erase_arg_t::SDMMC_ERASE_ARG (C++ enumerator), 1766
sdmmc_erase_sectors (C++ function), 1756
sdmmc_ext_csd_t (C++ struct), 1762
sdmmc_ext_csd_t::erase_mem_state (C++ member), 1762
sdmmc_ext_csd_t::power_class (C++ member), 1762
sdmmc_ext_csd_t::rev (C++ member), 1762
sdmmc_ext_csd_t::sec_feature (C++ member), 1762
SDMMC_FREQ_26M (C macro), 1766
SDMMC_FREQ_52M (C macro), 1766
SDMMC_FREQ_DEFAULT (C macro), 1766
SDMMC_FREQ_HIGHSPEED (C macro), 1766
SDMMC_FREQ_PROBING (C macro), 1766
sdmmc_full_erase (C++ function), 1757
sdmmc_get_status (C++ function), 1755
SDMMC_HOST_DEFAULT (C macro), 1273
sdmmc_host_deinit (C++ function), 1272
sdmmc_host_do_transaction (C++ function), 1272
SDMMC_HOST_FLAG_1BIT (C macro), 1765
SDMMC_HOST_FLAG_4BIT (C macro), 1765
SDMMC_HOST_FLAG_8BIT (C macro), 1766
SDMMC_HOST_FLAG_DDR (C macro), 1766
SDMMC_HOST_FLAG_DEINIT_ARG (C macro), 1766
SDMMC_HOST_FLAG_SPI (C macro), 1766
sdmmc_host_get_slot_width (C++ function), 1271
sdmmc_host_init (C++ function), 1270
sdmmc_host_init_slot (C++ function), 1270
sdmmc_host_io_int_enable (C++ function), 1272
sdmmc_host_io_int_wait (C++ function), 1272
sdmmc_host_set_bus_ddr_mode (C++ function), 1271
sdmmc_host_set_bus_width (C++ function), 1271
sdmmc_host_set_card_clk (C++ function), 1271
sdmmc_host_set_cclk_always_on (C++ function), 1272
SDMMC_HOST_SLOT_0 (C macro), 1273
SDMMC_HOST_SLOT_1 (C macro), 1273
sdmmc_host_t (C++ struct), 1763

sdmmc_host_t::command_timeout_ms (C++ member), 1764
sdmmc_host_t::deinit (C++ member), 1764
sdmmc_host_t::deinit_p (C++ member), 1764
sdmmc_host_t::do_transaction (C++ member), 1764
sdmmc_host_t::flags (C++ member), 1763
sdmmc_host_t::get_bus_width (C++ member), 1764
sdmmc_host_t::init (C++ member), 1763
sdmmc_host_t::io_int_enable (C++ member), 1764
sdmmc_host_t::io_int_wait (C++ member), 1764
sdmmc_host_t::io_voltage (C++ member), 1763
sdmmc_host_t::max_freq_khz (C++ member), 1763
sdmmc_host_t::set_bus_ddr_mode (C++ member), 1764
sdmmc_host_t::set_bus_width (C++ member), 1764
sdmmc_host_t::set_card_clk (C++ member), 1764
sdmmc_host_t::set_cclk_always_on (C++ member), 1764
sdmmc_host_t::slot (C++ member), 1763
sdmmc_io_enable_int (C++ function), 1759
sdmmc_io_get_cis_data (C++ function), 1759
sdmmc_io_print_cis_info (C++ function), 1759
sdmmc_io_read_blocks (C++ function), 1758
sdmmc_io_read_byte (C++ function), 1757
sdmmc_io_read_bytes (C++ function), 1758
sdmmc_io_wait_int (C++ function), 1759
sdmmc_io_write_blocks (C++ function), 1758
sdmmc_io_write_byte (C++ function), 1757
sdmmc_io_write_bytes (C++ function), 1758
sdmmc_mmc_can_sanitize (C++ function), 1757
sdmmc_mmc_sanitize (C++ function), 1757
sdmmc_read_sectors (C++ function), 1756
sdmmc_response_t (C++ type), 1766
sdmmc_scr_t (C++ struct), 1761
sdmmc_scr_t::bus_width (C++ member), 1761
sdmmc_scr_t::erase_mem_state (C++ member), 1761
sdmmc_scr_t::reserved (C++ member), 1761
sdmmc_scr_t::rsvd_mnf (C++ member), 1761
sdmmc_scr_t::sd_spec (C++ member), 1761
SDMMC_SLOT_CONFIG_DEFAULT (C macro), 1274
sdmmc_slot_config_t (C++ struct), 1273
sdmmc_slot_config_t::cd (C++ member), 1273
sdmmc_slot_config_t::flags (C++ member), 1273
sdmmc_slot_config_t::gpio_cd (C++ member), 1273
sdmmc_slot_config_t::gpio_wp (C++ member), 1273
sdmmc_slot_config_t::width (C++ member), 1273
sdmmc_slot_config_t::wp (C++ member), 1273
SDMMC_SLOT_FLAG_INTERNAL_PULLUP (C macro), 1273
SDMMC_SLOT_NO_CD (C macro), 1273
SDMMC_SLOT_NO_WP (C macro), 1273
SDMMC_SLOT_WIDTH_DEFAULT (C macro), 1273
sdmmc_ssr_t (C++ struct), 1761
sdmmc_ssr_t::alloc_unit_kb (C++ member), 1761
sdmmc_ssr_t::cur_bus_width (C++ member), 1762
sdmmc_ssr_t::discard_support (C++ member), 1762
sdmmc_ssr_t::erase_offset (C++ member), 1762
sdmmc_ssr_t::erase_size_au (C++ member), 1761
sdmmc_ssr_t::erase_timeout (C++ member), 1762
sdmmc_ssr_t::fule_support (C++ member), 1762
sdmmc_ssr_t::reserved (C++ member), 1762
sdmmc_switch_func_rsp_t (C++ struct), 1762
sdmmc_switch_func_rsp_t::data (C++ member), 1762
sdmmc_write_sectors (C++ function), 1756
SDP_OPP_SUPPORTED_FORMATS_MAX_LENGTH (C macro), 520
SDSPI_DEFAULT_DMA (C macro), 1278
SDSPI_DEFAULT_HOST (C macro), 1278
sdspi_dev_handle_t (C++ type), 1279
SDSPI_DEVICE_CONFIG_DEFAULT (C macro), 1279
sdspi_device_config_t (C++ struct), 1278
sdspi_device_config_t::gpio_cd (C++ member), 1278
sdspi_device_config_t::gpio_cs (C++ member), 1278
sdspi_device_config_t::gpio_int (C++ member), 1278
sdspi_device_config_t::gpio_wp (C++ member), 1278
sdspi_device_config_t::host_id (C++ member), 1278
SDSPI_HOST_DEFAULT (C macro), 1278
sdspi_host_deinit (C++ function), 1277
sdspi_host_do_transaction (C++ function), 1277
sdspi_host_init (C++ function), 1276
sdspi_host_init_device (C++ function), 1276
sdspi_host_io_int_enable (C++ function), 1277
sdspi_host_io_int_wait (C++ function), 1278

sdspi_host_remove_device (*C++ function*), 1277
sdspi_host_set_card_clk (*C++ function*), 1277
SDSPI_SLOT_NO_CD (*C macro*), 1278
SDSPI_SLOT_NO_CS (*C macro*), 1278
SDSPI_SLOT_NO_INT (*C macro*), 1279
SDSPI_SLOT_NO_WP (*C macro*), 1278
SemaphoreHandle_t (*C++ type*), 1967
semBINARY_SEMAPHORE_QUEUE_LENGTH (*C macro*), 1954
semGIVE_BLOCK_TIME (*C macro*), 1954
semSEMAPHORE_QUEUE_ITEM_LENGTH (*C macro*), 1954
shared_stack_function (*C++ type*), 1837
shutdown_handler_t (*C++ type*), 2085
slave_transaction_cb_t (*C++ type*), 1326
smartconfig_event_got_ssid_pswd_t (*C++ struct*), 872
smartconfig_event_got_ssid_pswd_t::bssid (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::bssid_set (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::cellphone_ip (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::password (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::ssid (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::token (*C++ member*), 872
smartconfig_event_got_ssid_pswd_t::type (*C++ member*), 872
smartconfig_event_t (*C++ enum*), 873
smartconfig_event_t::SC_EVENT_FOUND_CHANNEL (*C++ enumerator*), 873
smartconfig_event_t::SC_EVENT_GOT_SSID_PSWD (*C++ enumerator*), 873
smartconfig_event_t::SC_EVENT_SCAN_DONE (*C++ enumerator*), 873
smartconfig_event_t::SC_EVENT_SEND_ACK_DONE (*C++ enumerator*), 873
SMARTCONFIG_START_CONFIG_DEFAULT (*C macro*), 872
smartconfig_start_config_t (*C++ struct*), 872
smartconfig_start_config_t::enable_log (*C++ member*), 872
smartconfig_start_config_t::esp_touch_v2_enable_crypt (*C++ member*), 872
smartconfig_start_config_t::esp_touch_v2_key (*C++ member*), 872
smartconfig_type_t (*C++ enum*), 873
smartconfig_type_t::SC_TYPE_AIRKISS (*C++ enumerator*), 873
smartconfig_type_t::SC_TYPE_ESPTOUCH (*C++ enumerator*), 873
smartconfig_type_t::SC_TYPE_ESPTOUCH_AIRKISS (*C++ enumerator*), 873
smartconfig_type_t::SC_TYPE_ESPTOUCH_V2 (*C++ enumerator*), 873
sntp_get_sync_interval (*C++ function*), 2147
sntp_get_sync_mode (*C++ function*), 2146
sntp_get_sync_status (*C++ function*), 2147
sntp_restart (*C++ function*), 2147
sntp_set_sync_interval (*C++ function*), 2147
sntp_set_sync_mode (*C++ function*), 2146
sntp_set_sync_status (*C++ function*), 2147
sntp_set_time_sync_notification_cb (*C++ function*), 2147
sntp_sync_mode_t (*C++ enum*), 2149
sntp_sync_mode_t::SNTP_SYNC_MODE_IMMED (*C++ enumerator*), 2149
sntp_sync_mode_t::SNTP_SYNC_MODE_SMOOTH (*C++ enumerator*), 2149
sntp_sync_status_t (*C++ enum*), 2149
sntp_sync_status_t::SNTP_SYNC_STATUS_COMPLETED (*C++ enumerator*), 2149
sntp_sync_status_t::SNTP_SYNC_STATUS_IN_PROGRESS (*C++ enumerator*), 2149
sntp_sync_status_t::SNTP_SYNC_STATUS_RESET (*C++ enumerator*), 2149
sntp_sync_time (*C++ function*), 2146
sntp_sync_time_cb_t (*C++ type*), 2148
SOC_ADC_ATTEN_NUM (*C macro*), 2136
SOC_ADC_CHANNEL_NUM (*C macro*), 2136
SOC_ADC_DIG_CTRL_SUPPORTED (*C macro*), 2136
SOC_ADC_DIG_SUPPORTED_UNIT (*C macro*), 2136
SOC_ADC_DIGI_CONTROLLER_NUM (*C macro*), 2136
SOC_ADC_DIGI_DATA_BYTES_PER_CONV (*C macro*), 2136
SOC_ADC_DIGI_MAX_BITWIDTH (*C macro*), 2136
SOC_ADC_DIGI_MIN_BITWIDTH (*C macro*), 2136
SOC_ADC_DIGI_RESULT_BYTES (*C macro*), 2136
SOC_ADC_DMA_SUPPORTED (*C macro*), 2136
SOC_ADC_MAX_CHANNEL_NUM (*C macro*), 2136
SOC_ADC_PATT_LEN_MAX (*C macro*), 2136
SOC_ADC_PERIPH_NUM (*C macro*), 2136
SOC_ADC_RTC_CTRL_SUPPORTED (*C macro*), 2136
SOC_ADC_RTC_MAX_BITWIDTH (*C macro*), 2136
SOC_ADC_RTC_MIN_BITWIDTH (*C macro*), 2136
SOC_ADC_SAMPLE_FREQ_THRES_HIGH (*C macro*), 2136
SOC_ADC_SAMPLE_FREQ_THRES_LOW (*C macro*), 2136
SOC_AES_SUPPORTED (*C macro*), 2134
SOC_AES_SUPPORT_AES_128 (*C macro*), 2142
SOC_AES_SUPPORT_AES_192 (*C macro*), 2142
SOC_AES_SUPPORT_AES_256 (*C macro*), 2142
SOC_AES_SUPPORTED (*C macro*), 2135
SOC_APLL_MAX_HZ (*C macro*), 2138
SOC_APLL_MIN_HZ (*C macro*), 2138
SOC_APLL_MULTIPLIER_OUT_MAX_HZ (*C macro*), 2138

SOC_APLL_MULTIPLIER_OUT_MIN_HZ (*C macro*), 2137
SOC_BLE_DEVICE_PRIVACY_SUPPORTED (*C macro*), 2143
SOC_BLE_MESH_SUPPORTED (*C macro*), 2143
SOC_BLE_SUPPORTED (*C macro*), 2143
SOC_BT_CLASSIC_SUPPORTED (*C macro*), 2143
SOC_BT_SUPPORTED (*C macro*), 2135
SOC_CAPS_ECO_VER_MAX (*C macro*), 2134
SOC_CCOMP_TIMER_SUPPORTED (*C macro*), 2135
SOC_CLK_APLL_SUPPORTED (*C macro*), 2137
SOC_CLK_RC_FAST_D256_FREQ_APPROX (*C macro*), 1042
SOC_CLK_RC_FAST_FREQ_APPROX (*C macro*), 1042
SOC_CLK_RC_SLOW_FREQ_APPROX (*C macro*), 1042
SOC_CLK_XTAL32K_FREQ_APPROX (*C macro*), 1043
SOC_CPU_BREAKPOINTS_NUM (*C macro*), 2137
soc_cpu_clk_src_t (*C++ enum*), 1044
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_APLL (*C++ enumerator*), 1044
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_INVALID (*C++ enumerator*), 1044
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_PLL (*C++ enumerator*), 1044
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_RC_FAST (*C++ enumerator*), 1044
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_XTAL (*C++ enumerator*), 1044
SOC_CPU_CORES_NUM (*C macro*), 2137
SOC_CPU_HAS_FPU (*C macro*), 2137
SOC_CPU_INTR_NUM (*C macro*), 2137
SOC_CPU_WATCHPOINT_MAX_REGION_SIZE (*C macro*), 2137
SOC_CPU_WATCHPOINTS_NUM (*C macro*), 2137
SOC_DAC_PERIPH_NUM (*C macro*), 2137
SOC_DAC_RESOLUTION (*C macro*), 2137
SOC_DAC_SUPPORTED (*C macro*), 2134
SOC_DPORT_WORKAROUND (*C macro*), 2135
SOC_DPORT_WORKAROUND_DIS_INTERRUPT_LVL (*C macro*), 2135
SOC_EFUSE_SECURE_BOOT_KEY_DIGESTS (*C macro*), 2142
SOC_EMAC_SUPPORTED (*C macro*), 2135
SOC_FLASH_ENC_SUPPORTED (*C macro*), 2135
SOC_FLASH_ENCRYPTED_XTS_AES_BLOCK_MAX (*C macro*), 2142
SOC_GPIO_PIN_COUNT (*C macro*), 2137
SOC_GPIO_PORT (*C macro*), 2137
SOC_GPIO_VALID_DIGITAL_IO_PAD_MASK (*C macro*), 2137
SOC_GPIO_VALID_GPIO_MASK (*C macro*), 2137
SOC_GPIO_VALID_OUTPUT_GPIO_MASK (*C macro*), 2137
SOC_GPTIMER_CLKS (*C macro*), 1043
SOC_I2C_CLKS (*C macro*), 1043
SOC_I2C_FIFO_LEN (*C macro*), 2137
SOC_I2C_NUM (*C macro*), 2137
SOC_I2C_SUPPORT_APB (*C macro*), 2137
SOC_I2C_SUPPORT_SLAVE (*C macro*), 2137
SOC_I2S_CLKS (*C macro*), 1043
SOC_I2S_HW_VERSION_1 (*C macro*), 2138
SOC_I2S_LCD_I80_VARIANT (*C macro*), 2138
SOC_I2S_NUM (*C macro*), 2138
SOC_I2S_SUPPORTED (*C macro*), 2135
SOC_I2S_SUPPORTS_ADC (*C macro*), 2138
SOC_I2S_SUPPORTS_ADC_DAC (*C macro*), 2138
SOC_I2S_SUPPORTS_APLL (*C macro*), 2138
SOC_I2S_SUPPORTS_DAC (*C macro*), 2138
SOC_I2S_SUPPORTS_LCD_CAMERA (*C macro*), 2138
SOC_I2S_SUPPORTS_PDM (*C macro*), 2138
SOC_I2S_SUPPORTS_PDM_RX (*C macro*), 2138
SOC_I2S_SUPPORTS_PDM_TX (*C macro*), 2138
SOC_I2S_TRANS_SIZE_ALIGN_WORD (*C macro*), 2138
SOC_IDCACHE_PER_CORE (*C macro*), 2137
SOC_LCD_CLKS (*C macro*), 1043
SOC_LCD_I80_BUS_WIDTH (*C macro*), 2138
SOC_LCD_I80_BUSES (*C macro*), 2138
SOC_LCD_I80_SUPPORTED (*C macro*), 2138
SOC_LEDC_CHANNEL_NUM (*C macro*), 2139
SOC_LEDC_HAS_TIMER_SPECIFIC_MUX (*C macro*), 2138
SOC_LEDC_SUPPORT_APB_CLOCK (*C macro*), 2138
SOC_LEDC_SUPPORT_HS_MODE (*C macro*), 2138
SOC_LEDC_SUPPORT_REF_TICK (*C macro*), 2138
SOC_LEDC_TIMER_BIT_WIDE_NUM (*C macro*), 2139
SOC_MCPWM_CAPTURE_CHANNELS_PER_TIMER (*C macro*), 2139
SOC_MCPWM_CAPTURE_CLKS (*C macro*), 1043
SOC_MCPWM_CAPTURE_TIMERS_PER_GROUP (*C macro*), 2139
SOC_MCPWM_CARRIER_CLKS (*C macro*), 1043
SOC_MCPWM_COMPARATORS_PER_OPERATOR (*C macro*), 2139
SOC_MCPWM_GENERATORS_PER_OPERATOR (*C macro*), 2139
SOC_MCPWM_GPIO_FAULTS_PER_GROUP (*C macro*), 2139
SOC_MCPWM_GPIO_SYNCHROS_PER_GROUP (*C macro*), 2139
SOC_MCPWM_GROUPS (*C macro*), 2139
SOC_MCPWM_OPERATORS_PER_GROUP (*C macro*), 2139
SOC_MCPWM_SUPPORTED (*C macro*), 2134
SOC_MCPWM_TIMER_CLKS (*C macro*), 1043
SOC_MCPWM_TIMERS_PER_GROUP (*C macro*), 2139
SOC_MCPWM_TRIGGERS_PER_OPERATOR (*C macro*), 2139
SOC_MEMSPI_SRC_FREQ_20M_SUPPORTED (*C macro*), 2141

SOC_MEMSPI_SRC_FREQ_26M_SUPPORTED (C macro), 2141
SOC_MEMSPI_SRC_FREQ_40M_SUPPORTED (C macro), 2141
SOC_MEMSPI_SRC_FREQ_80M_SUPPORTED (C macro), 2141
SOC_MMU_LINEAR_ADDRESS_REGION_NUM (C macro), 2137
soc_module_clk_t (C++ enum), 1045
soc_module_clk_t::SOC_MOD_CLK_APB (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_APLL (C++ enumerator), 1046
soc_module_clk_t::SOC_MOD_CLK_CPU (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_PLL_D2 (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_PLL_F160M (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_RC_FAST (C++ enumerator), 1046
soc_module_clk_t::SOC_MOD_CLK_RC_FAST_D256 (C++ enumerator), 1046
soc_module_clk_t::SOC_MOD_CLK_REF_TICK (C++ enumerator), 1046
soc_module_clk_t::SOC_MOD_CLK_RTC_FAST (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_RTC_SLOW (C++ enumerator), 1045
soc_module_clk_t::SOC_MOD_CLK_XTAL (C++ enumerator), 1046
soc_module_clk_t::SOC_MOD_CLK_XTAL32K (C++ enumerator), 1045
SOC_MPI_SUPPORTED (C macro), 2135
SOC_MPU_CONFIGURABLE_REGIONS_SUPPORTED (C macro), 2139
SOC_MPU_MIN_REGION_SIZE (C macro), 2139
SOC_MPU_REGION_RO_SUPPORTED (C macro), 2139
SOC_MPU_REGION_WO_SUPPORTED (C macro), 2139
SOC_MPU_REGIONS_MAX_NUM (C macro), 2139
SOC_PCNT_CHANNELS_PER_UNIT (C macro), 2140
SOC_PCNT_GROUPS (C macro), 2139
SOC_PCNT_SUPPORTED (C macro), 2135
SOC_PCNT_THRES_POINT_PER_UNIT (C macro), 2140
SOC_PCNT_UNITS_PER_GROUP (C macro), 2139
soc_periph_gptimer_clk_src_t (C++ enum), 1046
soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_APB (C++ enumerator), 1046
soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_DEFAULT (C++ enumerator), 1046
soc_periph_i2c_clk_src_t (C++ enum), 1048
soc_periph_i2c_clk_src_t::I2C_CLK_SRC_APB (C++ enumerator), 1048
soc_periph_i2c_clk_src_t::I2C_CLK_SRC_DEFAULT (C++ enumerator), 1048
soc_periph_i2s_clk_src_t (C++ enum), 1048
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_APLL (C++ enumerator), 1048
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_DEFAULT (C++ enumerator), 1048
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_PLL_160M (C++ enumerator), 1048
soc_periph_lcd_clk_src_t (C++ enum), 1046
soc_periph_lcd_clk_src_t::LCD_CLK_SRC_DEFAULT (C++ enumerator), 1046
soc_periph_lcd_clk_src_t::LCD_CLK_SRC_PLL160M (C++ enumerator), 1046
soc_periph_mcpwm_capture_clk_src_t (C++ enum), 1047
soc_periph_mcpwm_capture_clk_src_t::MCPWM_CAPTURE (C++ enumerator), 1048
soc_periph_mcpwm_capture_clk_src_t::MCPWM_CAPTURE (C++ enumerator), 1048
soc_periph_mcpwm_carrier_clk_src_t (C++ enum), 1048
soc_periph_mcpwm_carrier_clk_src_t::MCPWM_CARRIER (C++ enumerator), 1048
soc_periph_mcpwm_carrier_clk_src_t::MCPWM_CARRIER (C++ enumerator), 1048
soc_periph_mcpwm_timer_clk_src_t (C++ enum), 1047
soc_periph_mcpwm_timer_clk_src_t::MCPWM_TIMER_CLK (C++ enumerator), 1047
soc_periph_mcpwm_timer_clk_src_t::MCPWM_TIMER_CLK (C++ enumerator), 1047
soc_periph_rmt_clk_src_legacy_t (C++ enum), 1047
soc_periph_rmt_clk_src_legacy_t::RMT_BASECLK_APB (C++ enumerator), 1047
soc_periph_rmt_clk_src_legacy_t::RMT_BASECLK_DEFA (C++ enumerator), 1047
soc_periph_rmt_clk_src_legacy_t::RMT_BASECLK_REF (C++ enumerator), 1047
soc_periph_rmt_clk_src_t (C++ enum), 1046
soc_periph_rmt_clk_src_t::RMT_CLK_SRC_APB (C++ enumerator), 1047
soc_periph_rmt_clk_src_t::RMT_CLK_SRC_DEFAULT (C++ enumerator), 1047
soc_periph_rmt_clk_src_t::RMT_CLK_SRC_REF_TICK (C++ enumerator), 1047
soc_periph_sdm_clk_src_t (C++ enum), 1048
soc_periph_sdm_clk_src_t::SDM_CLK_SRC_APB (C++ enumerator), 1048
soc_periph_sdm_clk_src_t::SDM_CLK_SRC_DEFAULT (C++ enumerator), 1048
soc_periph_tg_clk_src_legacy_t (C++ enum), 1046
soc_periph_tg_clk_src_legacy_t::TIMER_SRC_CLK_APB (C++ enumerator), 1046
soc_periph_tg_clk_src_legacy_t::TIMER_SRC_CLK_DEF (C++ enumerator), 1046
soc_periph_uart_clk_src_legacy_t (C++

enum), 1047
soc_periph_uart_clk_src_legacy_t::UART_SCLK_APB (*C++ enumerator*), 1047
soc_periph_uart_clk_src_legacy_t::UART_SCLK_DEFAULT (*C++ enumerator*), 1047
soc_periph_uart_clk_src_legacy_t::UART_SCLK_RTC (*C++ enumerator*), 1047
SOC_PHY_DIG_REGS_MEM_SIZE (*C macro*), 2142
SOC_PM_SUPPORT_EXT_WAKEUP (*C macro*), 2142
SOC_PM_SUPPORT_MODEM_PD (*C macro*), 2143
SOC_PM_SUPPORT_RTC_FAST_MEM_PD (*C macro*), 2143
SOC_PM_SUPPORT_RTC_PERIPH_PD (*C macro*), 2143
SOC_PM_SUPPORT_RTC_SLOW_MEM_PD (*C macro*), 2143
SOC_PM_SUPPORT_TOUCH_SENSOR_WAKEUP (*C macro*), 2142
SOC_REG_TO_ULP_PERIPH_SEL (*C++ function*), 2166
SOC_RMT_CHANNEL_CLK_INDEPENDENT (*C macro*), 2140
SOC_RMT_CHANNELS_PER_GROUP (*C macro*), 2140
SOC_RMT_CLKS (*C macro*), 1043
SOC_RMT_GROUPS (*C macro*), 2140
SOC_RMT_MEM_WORDS_PER_CHANNEL (*C macro*), 2140
SOC_RMT_RX_CANDIDATES_PER_GROUP (*C macro*), 2140
SOC_RMT_SUPPORT_APB (*C macro*), 2140
SOC_RMT_SUPPORT_REF_TICK (*C macro*), 2140
SOC_RMT_SUPPORTED (*C macro*), 2135
SOC_RMT_TX_CANDIDATES_PER_GROUP (*C macro*), 2140
soc_root_clk_t (*C++ enum*), 1043
soc_root_clk_t::SOC_ROOT_CLK_EXT_XTAL (*C++ enumerator*), 1044
soc_root_clk_t::SOC_ROOT_CLK_EXT_XTAL32K (*C++ enumerator*), 1044
soc_root_clk_t::SOC_ROOT_CLK_INT_RC_FAST (*C++ enumerator*), 1043
soc_root_clk_t::SOC_ROOT_CLK_INT_RC_SLOW (*C++ enumerator*), 1043
SOC_RSA_MAX_BIT_LEN (*C macro*), 2142
soc_rtc_fast_clk_src_t (*C++ enum*), 1045
soc_rtc_fast_clk_src_t::SOC_RTC_FAST_CLK_SRC_INVALID (*C++ enumerator*), 1045
soc_rtc_fast_clk_src_t::SOC_RTC_FAST_CLK_SRC_RC_FAST (*C++ enumerator*), 1045
soc_rtc_fast_clk_src_t::SOC_RTC_FAST_CLK_SRC_XTAL_D4 (*C++ enumerator*), 1045
soc_rtc_fast_clk_src_t::SOC_RTC_FAST_CLK_SRC_XTAL_DIV (*C++ enumerator*), 1045
SOC_RTC_FAST_MEM_SUPPORTED (*C macro*), 2135
SOC_RTC_MEM_SUPPORTED (*C macro*), 2135
soc_rtc_slow_clk_src_t (*C++ enum*), 1044
soc_rtc_slow_clk_src_t::SOC_RTC_SLOW_CLK_SRC_INVALID (*C++ enumerator*), 1044
soc_rtc_slow_clk_src_t::SOC_RTC_SLOW_CLK_SRC_RC_FAST (*C++ enumerator*), 1044
soc_rtc_slow_clk_src_t::SOC_RTC_SLOW_CLK_SRC_RC_SLOW (*C++ enumerator*), 1044
soc_rtc_slow_clk_src_t::SOC_RTC_SLOW_CLK_SRC_XTAL32K (*C++ enumerator*), 1044
SOC_RTC_SLOW_CLOCK_SUPPORT_8MD256 (*C macro*), 2136
SOC_RTC_SLOW_MEM_SUPPORTED (*C macro*), 2135
SOC_RTCIO_HOLD_SUPPORTED (*C macro*), 2140
SOC_RTCIO_INPUT_OUTPUT_SUPPORTED (*C macro*), 2140
SOC_RTCIO_PIN_COUNT (*C macro*), 2140
SOC_RTCIO_WAKE_SUPPORTED (*C macro*), 2140
SOC_SDIO_SLAVE_SUPPORTED (*C macro*), 2135
SOC_SDM_CHANNELS_PER_GROUP (*C macro*), 2140
SOC_SDM_CLKS (*C macro*), 1043
SOC_SDM_GROUPS (*C macro*), 2140
SOC_SDM_SUPPORTED (*C macro*), 2135
SOC_SDMMC_HOST_SUPPORTED (*C macro*), 2134
SOC_SDMMC_NUM_SLOTS (*C macro*), 2143
SOC_SDMMC_USE_IOMUX (*C macro*), 2143
SOC_SECURE_BOOT_SUPPORTED (*C macro*), 2135
SOC_SECURE_BOOT_V1 (*C macro*), 2142
SOC_SHA_SUPPORT_PARALLEL_ENG (*C macro*), 2142
SOC_SHA_SUPPORT_SHA1 (*C macro*), 2142
SOC_SHA_SUPPORT_SHA256 (*C macro*), 2142
SOC_SHA_SUPPORT_SHA384 (*C macro*), 2142
SOC_SHA_SUPPORT_SHA512 (*C macro*), 2142
SOC_SHA_SUPPORTED (*C macro*), 2135
SOC_SHARED_IDCACHE_SUPPORTED (*C macro*), 2137
SOC_SPI_AS_CS_SUPPORTED (*C macro*), 2140
SOC_SPI_DMA_CHAN_NUM (*C macro*), 2140
SOC_SPI_HD_BOTH_INOUT_SUPPORTED (*C macro*), 2140
SOC_SPI_MAX_CS_NUM (*C macro*), 2141
SOC_SPI_MAX_PRE_DIVIDER (*C macro*), 2141
SOC_SPI_MAXIMUM_BUFFER_SIZE (*C macro*), 2141
SOC_SPI_MEM_SUPPORT_CONFIG_GPIO_BY_EFUSE (*C macro*), 2142
SOC_SPI_PERIPH_CS_NUM (*C macro*), 2141
SOC_SPI_PERIPH_NUM (*C macro*), 2140
SOC_SPI_PERIPH_SUPPORT_MULTILINE_MODE (*C macro*), 2141
SOC_SPI_SUPPORT_CLK_RC_FAST (*C macro*), 2142
SOC_SUPPORT_COEXISTENCE (*C macro*), 2135
SOC_SYSTIMER_BIT_WIDTH (*C macro*), 2141
SOC_SYSTIMER_COUNTER_NUM (*C macro*), 2141
SOC_SYSTIMER_SUPPORT_APB (*C macro*), 2141
SOC_TIMER_GROUP_TIMERS_PER_GROUP (*C macro*), 2141
SOC_TIMER_GROUP_TOTAL_TIMERS (*C macro*), 2141
SOC_TIMER_GROUPS (*C macro*), 2141

SOC_TOUCH_PAD_MEASURE_WAIT_MAX (*C macro*), 2141
SOC_TOUCH_PAD_THRESHOLD_MAX (*C macro*), 2141
SOC_TOUCH_SENSOR_NUM (*C macro*), 2141
SOC_TOUCH_SENSOR_SUPPORTED (*C macro*), 2135
SOC_TOUCH_VERSION_1 (*C macro*), 2141
SOC_TWAI_BRP_MAX (*C macro*), 2141
SOC_TWAI_BRP_MIN (*C macro*), 2141
SOC_TWAI_SUPPORT_MULTI_ADDRESS_LAYOUT (*C macro*), 2141
SOC_TWAI_SUPPORTED (*C macro*), 2135
SOC_UART_BITRATE_MAX (*C macro*), 2142
SOC_UART_FIFO_LEN (*C macro*), 2142
SOC_UART_NUM (*C macro*), 2141
SOC_UART_SUPPORT_APB_CLK (*C macro*), 2142
SOC_UART_SUPPORT_REF_TICK (*C macro*), 2142
SOC_ULP_SUPPORTED (*C macro*), 2135
SOC_WIFI_CSI_SUPPORT (*C macro*), 2143
SOC_WIFI_FTM_SUPPORT (*C macro*), 2143
SOC_WIFI_GCMP_SUPPORT (*C macro*), 2143
SOC_WIFI_HW_TSF (*C macro*), 2143
SOC_WIFI_MESH_SUPPORT (*C macro*), 2143
SOC_WIFI_SUPPORTED (*C macro*), 2135
SOC_WIFI_WAPI_SUPPORT (*C macro*), 2143
SOC_XTAL_SUPPORT_26M (*C macro*), 2136
SOC_XTAL_SUPPORT_40M (*C macro*), 2136
SOC_XTAL_SUPPORT_AUTO_DETECT (*C macro*), 2136
spi_bus_add_device (*C++ function*), 1310
spi_bus_add_flash_device (*C++ function*), 1775
spi_bus_config_t (*C++ struct*), 1307
spi_bus_config_t::data0_io_num (*C++ member*), 1307
spi_bus_config_t::data1_io_num (*C++ member*), 1307
spi_bus_config_t::data2_io_num (*C++ member*), 1308
spi_bus_config_t::data3_io_num (*C++ member*), 1308
spi_bus_config_t::data4_io_num (*C++ member*), 1308
spi_bus_config_t::data5_io_num (*C++ member*), 1308
spi_bus_config_t::data6_io_num (*C++ member*), 1308
spi_bus_config_t::data7_io_num (*C++ member*), 1308
spi_bus_config_t::flags (*C++ member*), 1308
spi_bus_config_t::intr_flags (*C++ member*), 1308
spi_bus_config_t::max_transfer_sz (*C++ member*), 1308
spi_bus_config_t::miso_io_num (*C++ member*), 1307
spi_bus_config_t::mosi_io_num (*C++ member*), 1307
spi_bus_config_t::quadhd_io_num (*C++ member*), 1308
spi_bus_config_t::quadwp_io_num (*C++ member*), 1308
spi_bus_config_t::sclk_io_num (*C++ member*), 1307
spi_bus_free (*C++ function*), 1307
spi_bus_get_max_transaction_len (*C++ function*), 1314
spi_bus_initialize (*C++ function*), 1306
spi_bus_remove_device (*C++ function*), 1310
spi_bus_remove_flash_device (*C++ function*), 1775
spi_clock_source_t (*C++ enum*), 1305
spi_clock_source_t::SPI_CLK_APB (*C++ enumerator*), 1305
spi_clock_source_t::SPI_CLK_XTAL (*C++ enumerator*), 1305
spi_command_t (*C++ enum*), 1305
spi_command_t::SPI_CMD_HD_EN_QPI (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_INT0 (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_INT1 (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_INT2 (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_RDBUF (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_RDDMA (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_SEG_END (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_WR_END (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_WRBUF (*C++ enumerator*), 1306
spi_command_t::SPI_CMD_HD_WRDMA (*C++ enumerator*), 1306
spi_common_dma_t (*C++ enum*), 1310
spi_common_dma_t::SPI_DMA_CH1 (*C++ enumerator*), 1310
spi_common_dma_t::SPI_DMA_CH2 (*C++ enumerator*), 1310
spi_common_dma_t::SPI_DMA_CH_AUTO (*C++ enumerator*), 1310
spi_common_dma_t::SPI_DMA_DISABLED (*C++ enumerator*), 1310
SPI_DEVICE_3WIRE (*C macro*), 1317
spi_device_acquire_bus (*C++ function*), 1313
SPI_DEVICE_BIT_LSBFIRST (*C macro*), 1317
SPI_DEVICE_CLK_AS_CS (*C macro*), 1317
SPI_DEVICE_DDRCLK (*C macro*), 1318
spi_device_get_trans_result (*C++ function*), 1311
SPI_DEVICE_HALFDUPLEX (*C macro*), 1317
spi_device_handle_t (*C++ type*), 1319

spi_device_interface_config_t (*C++ struct*), 1314
spi_device_interface_config_t::address_bits (*C++ member*), 1314
spi_device_interface_config_t::clock_speed_hz (*C++ member*), 1315
spi_device_interface_config_t::command_bits (*C++ member*), 1314
spi_device_interface_config_t::cs_ena_posttrans (*C++ member*), 1314
spi_device_interface_config_t::cs_ena_pretrans (*C++ member*), 1314
spi_device_interface_config_t::dummy_bits (*C++ member*), 1314
spi_device_interface_config_t::duty_cycle_pos (*C++ member*), 1314
spi_device_interface_config_t::flags (*C++ member*), 1315
spi_device_interface_config_t::input_delay_ns (*C++ member*), 1315
spi_device_interface_config_t::mode (*C++ member*), 1314
spi_device_interface_config_t::post_cb (*C++ member*), 1315
spi_device_interface_config_t::pre_cb (*C++ member*), 1315
spi_device_interface_config_t::queue_size (*C++ member*), 1315
spi_device_interface_config_t::spics_io_num (*C++ member*), 1315
SPI_DEVICE_NO_DUMMY (*C macro*), 1318
spi_device_polling_end (*C++ function*), 1312
spi_device_polling_start (*C++ function*), 1312
spi_device_polling_transmit (*C++ function*), 1312
SPI_DEVICE_POSITIVE_CS (*C macro*), 1317
spi_device_queue_trans (*C++ function*), 1311
spi_device_release_bus (*C++ function*), 1313
SPI_DEVICE_RXBIT_LSBFIRST (*C macro*), 1317
spi_device_transmit (*C++ function*), 1311
SPI_DEVICE_TXBIT_LSBFIRST (*C macro*), 1317
spi_dma_chan_t (*C++ type*), 1310
spi_event_t (*C++ enum*), 1305
spi_event_t::SPI_EV_BUF_RX (*C++ enumerator*), 1305
spi_event_t::SPI_EV_BUF_TX (*C++ enumerator*), 1305
spi_event_t::SPI_EV_CMD9 (*C++ enumerator*), 1305
spi_event_t::SPI_EV_CMDA (*C++ enumerator*), 1305
spi_event_t::SPI_EV_RECV (*C++ enumerator*), 1305
spi_event_t::SPI_EV_RECV_DMA_READY (*C++ enumerator*), 1305
spi_event_t::SPI_EV_SEND (*C++ enumerator*), 1305
spi_event_t::SPI_EV_SEND_DMA_READY (*C++ enumerator*), 1305
spi_event_t::SPI_EV_TRANS (*C++ enumerator*), 1305
spi_flash_cache2phys (*C++ function*), 1785
SPI_FLASH_CACHE2PHYS_FAIL (*C macro*), 1785
spi_flash_chip_t (*C++ type*), 1783
SPI_FLASH_CONFIG_CONF_BITS (*C macro*), 1790
spi_flash_encryption_t (*C++ struct*), 1787
spi_flash_encryption_t::flash_encryption_check (*C++ member*), 1788
spi_flash_encryption_t::flash_encryption_data_pre (*C++ member*), 1787
spi_flash_encryption_t::flash_encryption_destroy (*C++ member*), 1788
spi_flash_encryption_t::flash_encryption_disable (*C++ member*), 1787
spi_flash_encryption_t::flash_encryption_done (*C++ member*), 1788
spi_flash_encryption_t::flash_encryption_enable (*C++ member*), 1787
spi_flash_host_driver_s (*C++ struct*), 1788
spi_flash_host_driver_s::check_suspend (*C++ member*), 1790
spi_flash_host_driver_s::common_command (*C++ member*), 1788
spi_flash_host_driver_s::configure_host_io_mode (*C++ member*), 1789
spi_flash_host_driver_s::dev_config (*C++ member*), 1788
spi_flash_host_driver_s::erase_block (*C++ member*), 1789
spi_flash_host_driver_s::erase_chip (*C++ member*), 1788
spi_flash_host_driver_s::erase_sector (*C++ member*), 1788
spi_flash_host_driver_s::flush_cache (*C++ member*), 1790
spi_flash_host_driver_s::host_status (*C++ member*), 1789
spi_flash_host_driver_s::poll_cmd_done (*C++ member*), 1790
spi_flash_host_driver_s::program_page (*C++ member*), 1789
spi_flash_host_driver_s::read (*C++ member*), 1789
spi_flash_host_driver_s::read_data_slicer (*C++ member*), 1789
spi_flash_host_driver_s::read_id (*C++ member*), 1788
spi_flash_host_driver_s::read_status (*C++ member*), 1789
spi_flash_host_driver_s::resume (*C++ member*), 1790
spi_flash_host_driver_s::set_write_protect (*C++ member*), 1789
spi_flash_host_driver_s::supports_direct_read (*C++ member*), 1789

spi_flash_host_driver_s::supports_direct_write (C++ member), 1789
spi_flash_host_driver_s::sus_setup (C++ member), 1790
spi_flash_host_driver_s::suspend (C++ member), 1790
spi_flash_host_driver_s::write_data_slicer (C++ member), 1789
spi_flash_host_driver_t (C++ type), 1790
spi_flash_host_inst_t (C++ struct), 1788
spi_flash_host_inst_t::driver (C++ member), 1788
spi_flash_mmap (C++ function), 1784
spi_flash_mmap_dump (C++ function), 1784
spi_flash_mmap_get_free_pages (C++ function), 1784
spi_flash_mmap_handle_t (C++ type), 1786
spi_flash_mmap_memory_t (C++ enum), 1786
spi_flash_mmap_memory_t::SPI_FLASH_MMAP_DATA (C++ enumerator), 1786
spi_flash_mmap_memory_t::SPI_FLASH_MMAP_INST (C++ enumerator), 1786
spi_flash_mmap_pages (C++ function), 1784
SPI_FLASH_MMU_PAGE_SIZE (C macro), 1785
spi_flash_munmap (C++ function), 1784
SPI_FLASH_OPI_FLAG (C macro), 1790
SPI_FLASH_OS_IS_ERASING_STATUS_FLAG (C macro), 1783
spi_flash_phys2cache (C++ function), 1785
SPI_FLASH_READ_MODE_MIN (C macro), 1790
SPI_FLASH_SEC_SIZE (C macro), 1785
spi_flash_sus_cmd_conf (C++ struct), 1787
spi_flash_sus_cmd_conf::cmd_rdsr (C++ member), 1787
spi_flash_sus_cmd_conf::res_cmd (C++ member), 1787
spi_flash_sus_cmd_conf::reserved (C++ member), 1787
spi_flash_sus_cmd_conf::sus_cmd (C++ member), 1787
spi_flash_sus_cmd_conf::sus_mask (C++ member), 1787
SPI_FLASH_TRANS_FLAG_BYTE_SWAP (C macro), 1790
SPI_FLASH_TRANS_FLAG_CMD16 (C macro), 1790
SPI_FLASH_TRANS_FLAG_IGNORE_BASEIO (C macro), 1790
spi_flash_trans_t (C++ struct), 1786
spi_flash_trans_t::address (C++ member), 1786
spi_flash_trans_t::address_bitlen (C++ member), 1786
spi_flash_trans_t::command (C++ member), 1787
spi_flash_trans_t::dummy_bitlen (C++ member), 1787
spi_flash_trans_t::flags (C++ member), 1787
spi_flash_trans_t::io_mode (C++ member), 1787
spi_flash_trans_t::miso_data (C++ member), 1786
spi_flash_trans_t::miso_len (C++ member), 1786
spi_flash_trans_t::mosi_data (C++ member), 1786
spi_flash_trans_t::mosi_len (C++ member), 1786
spi_flash_trans_t::reserved (C++ member), 1786
SPI_FLASH_YIELD_REQ_SUSPEND (C macro), 1783
SPI_FLASH_YIELD_REQ_YIELD (C macro), 1783
SPI_FLASH_YIELD_STA_RESUME (C macro), 1783
spi_get_actual_clock (C++ function), 1313
spi_get_freq_limit (C++ function), 1314
spi_get_timing (C++ function), 1313
spi_host_device_t (C++ enum), 1304
spi_host_device_t::SPI1_HOST (C++ enumerator), 1304
spi_host_device_t::SPI2_HOST (C++ enumerator), 1304
spi_host_device_t::SPI3_HOST (C++ enumerator), 1305
spi_host_device_t::SPI_HOST_MAX (C++ enumerator), 1305
spi_line_mode_t (C++ struct), 1304
spi_line_mode_t::addr_lines (C++ member), 1304
spi_line_mode_t::cmd_lines (C++ member), 1304
spi_line_mode_t::data_lines (C++ member), 1304
SPI_MASTER_FREQ_10M (C macro), 1317
SPI_MASTER_FREQ_11M (C macro), 1317
SPI_MASTER_FREQ_13M (C macro), 1317
SPI_MASTER_FREQ_16M (C macro), 1317
SPI_MASTER_FREQ_20M (C macro), 1317
SPI_MASTER_FREQ_26M (C macro), 1317
SPI_MASTER_FREQ_40M (C macro), 1317
SPI_MASTER_FREQ_80M (C macro), 1317
SPI_MASTER_FREQ_8M (C macro), 1317
SPI_MASTER_FREQ_9M (C macro), 1317
SPI_MAX_DMA_LEN (C macro), 1308
SPI_SLAVE_BIT_LSBFIRST (C macro), 1326
spi_slave_free (C++ function), 1324
spi_slave_get_trans_result (C++ function), 1324
spi_slave_initialize (C++ function), 1323
spi_slave_interface_config_t (C++ struct), 1325
spi_slave_interface_config_t::flags (C++ member), 1325
spi_slave_interface_config_t::mode (C++ member), 1325
spi_slave_interface_config_t::post_setup_cb

(C++ *member*), 1325
spi_slave_interface_config_t::post_trans_cb (C++ *member*), 1325
spi_slave_interface_config_t::queue_size (C++ *member*), 1325
spi_slave_interface_config_t::spics_io_num (C++ *member*), 1325
spi_slave_queue_trans (C++ *function*), 1324
SPI_SLAVE_RXBIT_LSBFIRST (C *macro*), 1326
spi_slave_transaction_t (C++ *struct*), 1325
spi_slave_transaction_t (C++ *type*), 1326
spi_slave_transaction_t::length (C++ *member*), 1326
spi_slave_transaction_t::rx_buffer (C++ *member*), 1326
spi_slave_transaction_t::trans_len (C++ *member*), 1326
spi_slave_transaction_t::tx_buffer (C++ *member*), 1326
spi_slave_transaction_t::user (C++ *member*), 1326
spi_slave_transmit (C++ *function*), 1324
SPI_SLAVE_TXBIT_LSBFIRST (C *macro*), 1326
SPI_SWAP_DATA_RX (C *macro*), 1309
SPI_SWAP_DATA_TX (C *macro*), 1308
SPI_TRANS_CS_KEEP_ACTIVE (C *macro*), 1318
SPI_TRANS_MODE_DIO (C *macro*), 1318
SPI_TRANS_MODE_DIOQIO_ADDR (C *macro*), 1318
SPI_TRANS_MODE_OCT (C *macro*), 1318
SPI_TRANS_MODE_QIO (C *macro*), 1318
SPI_TRANS_MULTILINE_ADDR (C *macro*), 1319
SPI_TRANS_MULTILINE_CMD (C *macro*), 1318
SPI_TRANS_USE_RXDATA (C *macro*), 1318
SPI_TRANS_USE_TXDATA (C *macro*), 1318
SPI_TRANS_VARIABLE_ADDR (C *macro*), 1318
SPI_TRANS_VARIABLE_CMD (C *macro*), 1318
SPI_TRANS_VARIABLE_DUMMY (C *macro*), 1318
spi_transaction_ext_t (C++ *struct*), 1316
spi_transaction_ext_t::address_bits (C++ *member*), 1316
spi_transaction_ext_t::base (C++ *member*), 1316
spi_transaction_ext_t::command_bits (C++ *member*), 1316
spi_transaction_ext_t::dummy_bits (C++ *member*), 1316
spi_transaction_t (C++ *struct*), 1315
spi_transaction_t (C++ *type*), 1319
spi_transaction_t::addr (C++ *member*), 1315
spi_transaction_t::cmd (C++ *member*), 1315
spi_transaction_t::flags (C++ *member*), 1315
spi_transaction_t::length (C++ *member*), 1316
spi_transaction_t::rx_buffer (C++ *member*), 1316
spi_transaction_t::rx_data (C++ *member*), 1316
spi_transaction_t::rxlength (C++ *member*), 1316
spi_transaction_t::tx_buffer (C++ *member*), 1316
spi_transaction_t::tx_data (C++ *member*), 1316
spi_transaction_t::user (C++ *member*), 1316
SPICOMMON_BUSFLAG_DUAL (C *macro*), 1309
SPICOMMON_BUSFLAG_GPIO_PINS (C *macro*), 1309
SPICOMMON_BUSFLAG_IO4_IO7 (C *macro*), 1309
SPICOMMON_BUSFLAG_IOMUX_PINS (C *macro*), 1309
SPICOMMON_BUSFLAG_MASTER (C *macro*), 1309
SPICOMMON_BUSFLAG_MISO (C *macro*), 1309
SPICOMMON_BUSFLAG_MOSI (C *macro*), 1309
SPICOMMON_BUSFLAG_NATIVE_PINS (C *macro*), 1309
SPICOMMON_BUSFLAG_OCTAL (C *macro*), 1309
SPICOMMON_BUSFLAG_QUAD (C *macro*), 1309
SPICOMMON_BUSFLAG_SCLK (C *macro*), 1309
SPICOMMON_BUSFLAG_SLAVE (C *macro*), 1309
SPICOMMON_BUSFLAG_WPHD (C *macro*), 1309
StaticRingbuffer_t (C++ *type*), 2028
StreamBufferHandle_t (C++ *type*), 2002
SUB_OPCODE_ALU_CNT (C *macro*), 2172
SUB_OPCODE_ALU_IMM (C *macro*), 2172
SUB_OPCODE_ALU_REG (C *macro*), 2172
SUB_OPCODE_B (C *macro*), 2173
SUB_OPCODE_BR (C *macro*), 2173
SUB_OPCODE_BS (C *macro*), 2173
SUB_OPCODE_BX (C *macro*), 2173
SUB_OPCODE_END (C *macro*), 2173
SUB_OPCODE_I2C_RD (C *macro*), 2171
SUB_OPCODE_I2C_WR (C *macro*), 2171
SUB_OPCODE_MACRO_BRANCH (C *macro*), 2174
SUB_OPCODE_MACRO_LABEL (C *macro*), 2174
SUB_OPCODE_MACRO_LABELPC (C *macro*), 2174
SUB_OPCODE_SLEEP (C *macro*), 2173
SUB_OPCODE_ST (C *macro*), 2172

T

taskDISABLE_INTERRUPTS (C *macro*), 1932
taskENABLE_INTERRUPTS (C *macro*), 1932
taskENTER_CRITICAL (C *macro*), 1931
taskENTER_CRITICAL_FROM_ISR (C *macro*), 1931
taskENTER_CRITICAL_ISR (C *macro*), 1931
taskEXIT_CRITICAL (C *macro*), 1931
taskEXIT_CRITICAL_FROM_ISR (C *macro*), 1932
taskEXIT_CRITICAL_ISR (C *macro*), 1932
TaskHandle_t (C++ *type*), 1934
TaskHookFunction_t (C++ *type*), 1934
taskSCHEDULER_NOT_STARTED (C *macro*), 1932
taskSCHEDULER_RUNNING (C *macro*), 1932
taskSCHEDULER_SUSPENDED (C *macro*), 1932

taskYIELD (*C macro*), 1931
TimerCallbackFunction_t (*C++ type*), 1985
TimerHandle_t (*C++ type*), 1985
tls_keep_alive_cfg (*C++ struct*), 148
tls_keep_alive_cfg::keep_alive_count (*C++ member*), 148
tls_keep_alive_cfg::keep_alive_enable (*C++ member*), 148
tls_keep_alive_cfg::keep_alive_idle (*C++ member*), 148
tls_keep_alive_cfg::keep_alive_interval (*C++ member*), 148
tls_keep_alive_cfg_t (*C++ type*), 151
TlsDeleteCallbackFunction_t (*C++ type*), 1934
tmrCOMMAND_CHANGE_PERIOD (*C macro*), 1976
tmrCOMMAND_CHANGE_PERIOD_FROM_ISR (*C macro*), 1976
tmrCOMMAND_DELETE (*C macro*), 1976
tmrCOMMAND_EXECUTE_CALLBACK (*C macro*), 1975
tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR (*C macro*), 1975
tmrCOMMAND_RESET (*C macro*), 1975
tmrCOMMAND_RESET_FROM_ISR (*C macro*), 1976
tmrCOMMAND_START (*C macro*), 1975
tmrCOMMAND_START_DONT_TRACE (*C macro*), 1975
tmrCOMMAND_START_FROM_ISR (*C macro*), 1976
tmrCOMMAND_STOP (*C macro*), 1975
tmrCOMMAND_STOP_FROM_ISR (*C macro*), 1976
tmrFIRST_FROM_ISR_COMMAND (*C macro*), 1976
touch_cnt_slope_t (*C++ enum*), 1342
touch_cnt_slope_t::TOUCH_PAD_SLOPE_0 (*C++ enumerator*), 1342
touch_cnt_slope_t::TOUCH_PAD_SLOPE_1 (*C++ enumerator*), 1342
touch_cnt_slope_t::TOUCH_PAD_SLOPE_2 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_3 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_4 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_5 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_6 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_7 (*C++ enumerator*), 1343
touch_cnt_slope_t::TOUCH_PAD_SLOPE_MAX (*C++ enumerator*), 1343
TOUCH_FSM_MODE_DEFAULT (*C macro*), 1340
touch_fsm_mode_t (*C++ enum*), 1343
touch_fsm_mode_t::TOUCH_FSM_MODE_MAX (*C++ enumerator*), 1343
touch_fsm_mode_t::TOUCH_FSM_MODE_SW (*C++ enumerator*), 1343
touch_fsm_mode_t::TOUCH_FSM_MODE_TIMER (*C++ enumerator*), 1343
touch_high_volt_t (*C++ enum*), 1341
touch_high_volt_t::TOUCH_HVOLT_2V4 (*C++ enumerator*), 1341
touch_high_volt_t::TOUCH_HVOLT_2V5 (*C++ enumerator*), 1341
touch_high_volt_t::TOUCH_HVOLT_2V6 (*C++ enumerator*), 1341
touch_high_volt_t::TOUCH_HVOLT_2V7 (*C++ enumerator*), 1341
touch_high_volt_t::TOUCH_HVOLT_KEEP (*C++ enumerator*), 1341
touch_high_volt_t::TOUCH_HVOLT_MAX (*C++ enumerator*), 1341
touch_low_volt_t (*C++ enum*), 1341
touch_low_volt_t::TOUCH_LVOLT_0V5 (*C++ enumerator*), 1342
touch_low_volt_t::TOUCH_LVOLT_0V6 (*C++ enumerator*), 1342
touch_low_volt_t::TOUCH_LVOLT_0V7 (*C++ enumerator*), 1342
touch_low_volt_t::TOUCH_LVOLT_0V8 (*C++ enumerator*), 1342
touch_low_volt_t::TOUCH_LVOLT_KEEP (*C++ enumerator*), 1342
touch_low_volt_t::TOUCH_LVOLT_MAX (*C++ enumerator*), 1342
TOUCH_PAD_ATTEN_VOLTAGE_THRESHOLD (*C macro*), 1340
TOUCH_PAD_BIT_MASK_ALL (*C macro*), 1340
TOUCH_PAD_BIT_MASK_MAX (*C macro*), 1340
touch_pad_clear_group_mask (*C++ function*), 1334
touch_pad_clear_status (*C++ function*), 1338
touch_pad_config (*C++ function*), 1330
touch_pad_deinit (*C++ function*), 1336
touch_pad_filter_delete (*C++ function*), 1335
touch_pad_filter_start (*C++ function*), 1335
touch_pad_filter_stop (*C++ function*), 1335
touch_pad_get_cnt_mode (*C++ function*), 1337
touch_pad_get_filter_period (*C++ function*), 1335
touch_pad_get_fsm_mode (*C++ function*), 1338
touch_pad_get_group_mask (*C++ function*), 1334
touch_pad_get_meas_time (*C++ function*), 1333
touch_pad_get_measurement_clock_cycles (*C++ function*), 1332
touch_pad_get_measurement_interval (*C++ function*), 1332
touch_pad_get_status (*C++ function*), 1338
touch_pad_get_thresh (*C++ function*), 1333
touch_pad_get_trigger_mode (*C++ function*), 1333
touch_pad_get_trigger_source (*C++ function*), 1334

touch_pad_get_voltage (C++ function), 1337
touch_pad_get_wakeup_status (C++ function), 1338
TOUCH_PAD_GPIO0_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO12_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO13_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO14_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO15_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO27_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO2_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO32_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO33_CHANNEL (C macro), 1339
TOUCH_PAD_GPIO4_CHANNEL (C macro), 1339
TOUCH_PAD_HIGH_VOLTAGE_THRESHOLD (C macro), 1340
TOUCH_PAD_IDLE_CH_CONNECT_DEFAULT (C macro), 1340
touch_pad_init (C++ function), 1336
touch_pad_intr_clear (C++ function), 1335
touch_pad_intr_disable (C++ function), 1335
touch_pad_intr_enable (C++ function), 1334
touch_pad_io_init (C++ function), 1336
touch_pad_isr_deregister (C++ function), 1337
touch_pad_isr_register (C++ function), 1331
TOUCH_PAD_LOW_VOLTAGE_THRESHOLD (C macro), 1340
touch_pad_meas_is_done (C++ function), 1338
TOUCH_PAD_MEASURE_CYCLE_DEFAULT (C macro), 1340
TOUCH_PAD_NUM0_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM1_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM2_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM3_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM4_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM5_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM6_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM7_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM8_GPIO_NUM (C macro), 1339
TOUCH_PAD_NUM9_GPIO_NUM (C macro), 1339
touch_pad_read (C++ function), 1330
touch_pad_read_filtered (C++ function), 1331
touch_pad_read_raw_data (C++ function), 1331
touch_pad_set_cnt_mode (C++ function), 1337
touch_pad_set_filter_period (C++ function), 1335
touch_pad_set_filter_read_cb (C++ function), 1331
touch_pad_set_fsm_mode (C++ function), 1338
touch_pad_set_group_mask (C++ function), 1334
touch_pad_set_meas_time (C++ function), 1332
touch_pad_set_measurement_clock_cycles (C++ function), 1332
touch_pad_set_measurement_interval (C++ function), 1332
touch_pad_set_thresh (C++ function), 1333
touch_pad_set_trigger_mode (C++ function), 1333
touch_pad_set_trigger_source (C++ function), 1334
touch_pad_set_voltage (C++ function), 1336
TOUCH_PAD_SLEEP_CYCLE_DEFAULT (C macro), 1340
TOUCH_PAD_SLOPE_DEFAULT (C macro), 1340
touch_pad_sw_start (C++ function), 1333
touch_pad_t (C++ enum), 1340
touch_pad_t::TOUCH_PAD_MAX (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM0 (C++ enumerator), 1340
touch_pad_t::TOUCH_PAD_NUM1 (C++ enumerator), 1340
touch_pad_t::TOUCH_PAD_NUM2 (C++ enumerator), 1340
touch_pad_t::TOUCH_PAD_NUM3 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM4 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM5 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM6 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM7 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM8 (C++ enumerator), 1341
touch_pad_t::TOUCH_PAD_NUM9 (C++ enumerator), 1341
TOUCH_PAD_THRESHOLD_MAX (C macro), 1340
TOUCH_PAD_TIE_OPT_DEFAULT (C macro), 1340
touch_tie_opt_t (C++ enum), 1343
touch_tie_opt_t::TOUCH_PAD_TIE_OPT_HIGH (C++ enumerator), 1343
touch_tie_opt_t::TOUCH_PAD_TIE_OPT_LOW (C++ enumerator), 1343
touch_tie_opt_t::TOUCH_PAD_TIE_OPT_MAX (C++ enumerator), 1343
TOUCH_TRIGGER_MODE_DEFAULT (C macro), 1340
touch_trigger_mode_t (C++ enum), 1343
touch_trigger_mode_t::TOUCH_TRIGGER_ABOVE (C++ enumerator), 1344
touch_trigger_mode_t::TOUCH_TRIGGER_BELOW (C++ enumerator), 1344
touch_trigger_mode_t::TOUCH_TRIGGER_MAX (C++ enumerator), 1344
TOUCH_TRIGGER_SOURCE_DEFAULT (C macro), 1340
touch_trigger_src_t (C++ enum), 1344
touch_trigger_src_t::TOUCH_TRIGGER_SOURCE_BOTH (C++ enumerator), 1344
touch_trigger_src_t::TOUCH_TRIGGER_SOURCE_MAX

(C++ *enumerator*), 1344
touch_trigger_src_t::TOUCH_TRIGGER_SOURCE_SET1 (C++ *enumerator*), 1344
touch_volt_atten_t (C++ *enum*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_0V (C++ *enumerator*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_0V5 (C++ *enumerator*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_1V (C++ *enumerator*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_1V5 (C++ *enumerator*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_KEEP (C++ *enumerator*), 1342
touch_volt_atten_t::TOUCH_HVOLT_ATTEN_MAX (C++ *enumerator*), 1342
transaction_cb_t (C++ *type*), 1319
tskDEFAULT_INDEX_TO_NOTIFY (C *macro*), 1931
tskIDLE_PRIORITY (C *macro*), 1931
tskKERNEL_VERSION_BUILD (C *macro*), 1931
tskKERNEL_VERSION_MAJOR (C *macro*), 1931
tskKERNEL_VERSION_MINOR (C *macro*), 1931
tskKERNEL_VERSION_NUMBER (C *macro*), 1931
tskMPU_REGION_DEVICE_MEMORY (C *macro*), 1931
tskMPU_REGION_EXECUTE_NEVER (C *macro*), 1931
tskMPU_REGION_NORMAL_MEMORY (C *macro*), 1931
tskMPU_REGION_READ_ONLY (C *macro*), 1931
tskMPU_REGION_READ_WRITE (C *macro*), 1931
tskNO_AFFINITY (C *macro*), 1931
twai_clear_receive_queue (C++ *function*), 1359
twai_clear_transmit_queue (C++ *function*), 1359
twai_driver_install (C++ *function*), 1356
twai_driver_uninstall (C++ *function*), 1356
TWAI_ERR_PASS_THRESH (C *macro*), 1356
TWAI_EXTD_ID_MASK (C *macro*), 1355
twai_filter_config_t (C++ *struct*), 1355
twai_filter_config_t::acceptance_code (C++ *member*), 1355
twai_filter_config_t::acceptance_mask (C++ *member*), 1355
twai_filter_config_t::single_filter (C++ *member*), 1355
TWAI_FRAME_EXTD_ID_LEN_BYTES (C *macro*), 1355
TWAI_FRAME_MAX_DLC (C *macro*), 1355
TWAI_FRAME_STD_ID_LEN_BYTES (C *macro*), 1355
twai_general_config_t (C++ *struct*), 1359
twai_general_config_t::alerts_enabled (C++ *member*), 1360
twai_general_config_t::bus_off_io (C++ *member*), 1360
twai_general_config_t::clkout_divider (C++ *member*), 1360
twai_general_config_t::clkout_io (C++ *member*), 1360
twai_general_config_t::intr_flags (C++ *member*), 1360
twai_general_config_t::mode (C++ *member*), 1360
twai_general_config_t::rx_io (C++ *member*), 1360
twai_general_config_t::rx_queue_len (C++ *member*), 1360
twai_general_config_t::tx_io (C++ *member*), 1360
twai_general_config_t::tx_queue_len (C++ *member*), 1360
twai_get_status_info (C++ *function*), 1359
twai_initiate_recovery (C++ *function*), 1359
TWAI_IO_UNUSED (C *macro*), 1361
twai_message_t (C++ *struct*), 1354
twai_message_t::data (C++ *member*), 1354
twai_message_t::data_length_code (C++ *member*), 1354
twai_message_t::dlc_non_comp (C++ *member*), 1354
twai_message_t::extd (C++ *member*), 1354
twai_message_t::flags (C++ *member*), 1354
twai_message_t::identifier (C++ *member*), 1354
twai_message_t::reserved (C++ *member*), 1354
twai_message_t::rtr (C++ *member*), 1354
twai_message_t::self (C++ *member*), 1354
twai_message_t::ss (C++ *member*), 1354
twai_mode_t (C++ *enum*), 1356
twai_mode_t::TWAI_MODE_LISTEN_ONLY (C++ *enumerator*), 1356
twai_mode_t::TWAI_MODE_NO_ACK (C++ *enumerator*), 1356
twai_mode_t::TWAI_MODE_NORMAL (C++ *enumerator*), 1356
twai_read_alerts (C++ *function*), 1358
twai_receive (C++ *function*), 1358
twai_reconfigure_alerts (C++ *function*), 1358
twai_start (C++ *function*), 1357
twai_state_t (C++ *enum*), 1361
twai_state_t::TWAI_STATE_BUS_OFF (C++ *enumerator*), 1361
twai_state_t::TWAI_STATE_RECOVERING (C++ *enumerator*), 1361
twai_state_t::TWAI_STATE_RUNNING (C++ *enumerator*), 1361
twai_state_t::TWAI_STATE_STOPPED (C++ *enumerator*), 1361
twai_status_info_t (C++ *struct*), 1360
twai_status_info_t::arb_lost_count (C++ *member*), 1361
twai_status_info_t::bus_error_count

(C++ *member*), 1361
twai_status_info_t::msgs_to_rx (C++ *member*), 1360
twai_status_info_t::msgs_to_tx (C++ *member*), 1360
twai_status_info_t::rx_error_counter (C++ *member*), 1361
twai_status_info_t::rx_missed_count (C++ *member*), 1361
twai_status_info_t::rx_overrun_count (C++ *member*), 1361
twai_status_info_t::state (C++ *member*), 1360
twai_status_info_t::tx_error_counter (C++ *member*), 1361
twai_status_info_t::tx_failed_count (C++ *member*), 1361
TWAI_STD_ID_MASK (C *macro*), 1355
twai_stop (C++ *function*), 1357
twai_timing_config_t (C++ *struct*), 1354
twai_timing_config_t::brp (C++ *member*), 1355
twai_timing_config_t::sjw (C++ *member*), 1355
twai_timing_config_t::triple_sampling (C++ *member*), 1355
twai_timing_config_t::tseg_1 (C++ *member*), 1355
twai_timing_config_t::tseg_2 (C++ *member*), 1355
twai_transmit (C++ *function*), 1357

U

uart_at_cmd_t (C++ *struct*), 1381
uart_at_cmd_t::char_num (C++ *member*), 1381
uart_at_cmd_t::cmd_char (C++ *member*), 1381
uart_at_cmd_t::gap_tout (C++ *member*), 1381
uart_at_cmd_t::post_idle (C++ *member*), 1381
uart_at_cmd_t::pre_idle (C++ *member*), 1381
UART_BITRATE_MAX (C *macro*), 1380
uart_clear_intr_status (C++ *function*), 1370
uart_config_t (C++ *struct*), 1382
uart_config_t::baud_rate (C++ *member*), 1382
uart_config_t::data_bits (C++ *member*), 1382
uart_config_t::flow_ctrl (C++ *member*), 1382
uart_config_t::parity (C++ *member*), 1382
uart_config_t::rx_flow_ctrl_thresh (C++ *member*), 1382
uart_config_t::source_clk (C++ *member*), 1382
uart_config_t::stop_bits (C++ *member*), 1382
UART_CTS_GPIO19_DIRECT_CHANNEL (C *macro*), 1386
UART_CTS_GPIO6_DIRECT_CHANNEL (C *macro*), 1386
UART_CTS_GPIO8_DIRECT_CHANNEL (C *macro*), 1387
uart_disable_intr_mask (C++ *function*), 1371
uart_disable_pattern_det_intr (C++ *function*), 1375
uart_disable_rx_intr (C++ *function*), 1371
uart_disable_tx_intr (C++ *function*), 1371
uart_driver_delete (C++ *function*), 1368
uart_driver_install (C++ *function*), 1368
uart_enable_intr_mask (C++ *function*), 1371
uart_enable_pattern_det_baud_intr (C++ *function*), 1375
uart_enable_rx_intr (C++ *function*), 1371
uart_enable_tx_intr (C++ *function*), 1371
uart_event_t (C++ *struct*), 1379
uart_event_t::size (C++ *member*), 1379
uart_event_t::timeout_flag (C++ *member*), 1379
uart_event_t::type (C++ *member*), 1379
uart_event_type_t (C++ *enum*), 1380
uart_event_type_t::UART_BREAK (C++ *enumerator*), 1380
uart_event_type_t::UART_BUFFER_FULL (C++ *enumerator*), 1380
uart_event_type_t::UART_DATA (C++ *enumerator*), 1380
uart_event_type_t::UART_DATA_BREAK (C++ *enumerator*), 1380
uart_event_type_t::UART_EVENT_MAX (C++ *enumerator*), 1381
uart_event_type_t::UART_FIFO_OVF (C++ *enumerator*), 1380
uart_event_type_t::UART_FRAME_ERR (C++ *enumerator*), 1380
uart_event_type_t::UART_PARITY_ERR (C++ *enumerator*), 1380
uart_event_type_t::UART_PATTERN_DET (C++ *enumerator*), 1381
UART_FIFO_LEN (C *macro*), 1380
uart_flush (C++ *function*), 1374
uart_flush_input (C++ *function*), 1374
uart_get_baudrate (C++ *function*), 1370
uart_get_buffered_data_len (C++ *function*), 1374
uart_get_collision_flag (C++ *function*), 1377
uart_get_hw_flow_ctrl (C++ *function*), 1370
uart_get_parity (C++ *function*), 1369
uart_get_sclk_freq (C++ *function*), 1369
uart_get_stop_bits (C++ *function*), 1369
uart_get_tx_buffer_free_size (C++ *function*), 1375

uart_get_wakeup_threshold (C++ function), 1378
uart_get_word_length (C++ function), 1368
UART_GPIO10_DIRECT_CHANNEL (C macro), 1386
UART_GPIO11_DIRECT_CHANNEL (C macro), 1386
UART_GPIO16_DIRECT_CHANNEL (C macro), 1386
UART_GPIO17_DIRECT_CHANNEL (C macro), 1386
UART_GPIO19_DIRECT_CHANNEL (C macro), 1385
UART_GPIO1_DIRECT_CHANNEL (C macro), 1385
UART_GPIO22_DIRECT_CHANNEL (C macro), 1385
UART_GPIO3_DIRECT_CHANNEL (C macro), 1385
UART_GPIO6_DIRECT_CHANNEL (C macro), 1386
UART_GPIO7_DIRECT_CHANNEL (C macro), 1387
UART_GPIO8_DIRECT_CHANNEL (C macro), 1386
UART_GPIO9_DIRECT_CHANNEL (C macro), 1386
uart_hw_flowcontrol_t (C++ enum), 1384
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_CTS (C++ enumerator), 1384
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_CTS_RTS (C++ enumerator), 1384
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_DISABLE (C++ enumerator), 1384
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_MAX (C++ enumerator), 1384
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_RTS (C++ enumerator), 1384
uart_intr_config (C++ function), 1373
uart_intr_config_t (C++ struct), 1379
uart_intr_config_t::intr_enable_mask (C++ member), 1379
uart_intr_config_t::rx_timeout_thresh (C++ member), 1379
uart_intr_config_t::rxfifo_full_thresh (C++ member), 1379
uart_intr_config_t::txfifo_empty_intr_thresh (C++ member), 1379
uart_is_driver_installed (C++ function), 1368
uart_isr_handle_t (C++ type), 1380
uart_mode_t (C++ enum), 1382
uart_mode_t::UART_MODE_IRDA (C++ enumerator), 1383
uart_mode_t::UART_MODE_RS485_APP_CTRL (C++ enumerator), 1383
uart_mode_t::UART_MODE_RS485_COLLISION_DETECT (C++ enumerator), 1383
uart_mode_t::UART_MODE_RS485_HALF_DUPLEX (C++ enumerator), 1383
uart_mode_t::UART_MODE_UART (C++ enumerator), 1382
UART_NUM_0 (C macro), 1380
UART_NUM_0_CTS_DIRECT_GPIO_NUM (C macro), 1385
UART_NUM_0_RTS_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_0_RXD_DIRECT_GPIO_NUM (C macro), 1385
UART_NUM_0_TXD_DIRECT_GPIO_NUM (C macro), 1385
UART_NUM_1 (C macro), 1380
UART_NUM_1_CTS_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_1_RTS_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_1_RXD_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_1_TXD_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_2 (C macro), 1380
UART_NUM_2_CTS_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_2_RTS_DIRECT_GPIO_NUM (C macro), 1387
UART_NUM_2_RXD_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_2_TXD_DIRECT_GPIO_NUM (C macro), 1386
UART_NUM_MAX (C macro), 1380
uart_param_config (C++ function), 1373
uart_parity_t (C++ enum), 1383
uart_parity_t::UART_PARITY_DISABLE (C++ enumerator), 1384
uart_parity_t::UART_PARITY_EVEN (C++ enumerator), 1384
uart_parity_t::UART_PARITY_ODD (C++ enumerator), 1384
uart_pattern_get_pos (C++ function), 1376
uart_pattern_pop_pos (C++ function), 1375
uart_pattern_queue_reset (C++ function), 1376
UART_PIN_NO_CHANGE (C macro), 1380
uart_port_t (C++ type), 1382
uart_read_bytes (C++ function), 1374
UART_RTS_GPIO11_DIRECT_CHANNEL (C macro), 1386
UART_RTS_GPIO22_DIRECT_CHANNEL (C macro), 1386
UART_RTS_GPIO7_DIRECT_CHANNEL (C macro), 1387
UART_RXD_GPIO16_DIRECT_CHANNEL (C macro), 1387
UART_RXD_GPIO3_DIRECT_CHANNEL (C macro), 1386
UART_RXD_GPIO9_DIRECT_CHANNEL (C macro), 1386
uart_sclk_t (C++ type), 1382
uart_set_always_rx_timeout (C++ function), 1379
uart_set_baudrate (C++ function), 1369
uart_set_dtr (C++ function), 1372
uart_set_hw_flow_ctrl (C++ function), 1370
uart_set_line_inverse (C++ function), 1370
uart_set_loop_back (C++ function), 1378
uart_set_mode (C++ function), 1376
uart_set_parity (C++ function), 1369
uart_set_pin (C++ function), 1371

uart_set_rts (C++ function), 1372
uart_set_rx_full_threshold (C++ function), 1376
uart_set_rx_timeout (C++ function), 1377
uart_set_stop_bits (C++ function), 1369
uart_set_sw_flow_ctrl (C++ function), 1370
uart_set_tx_empty_threshold (C++ function), 1377
uart_set_tx_idle_num (C++ function), 1372
uart_set_wakeup_threshold (C++ function), 1377
uart_set_word_length (C++ function), 1368
uart_signal_inv_t (C++ enum), 1384
uart_signal_inv_t::UART_SIGNAL_CTS_INV (C++ enumerator), 1384
uart_signal_inv_t::UART_SIGNAL_DSR_INV (C++ enumerator), 1385
uart_signal_inv_t::UART_SIGNAL_DTR_INV (C++ enumerator), 1385
uart_signal_inv_t::UART_SIGNAL_INV_DISABLE (C++ enumerator), 1384
uart_signal_inv_t::UART_SIGNAL_IRDA_RX_INV (C++ enumerator), 1384
uart_signal_inv_t::UART_SIGNAL_IRDA_TX_INV (C++ enumerator), 1384
uart_signal_inv_t::UART_SIGNAL_RTS_INV (C++ enumerator), 1385
uart_signal_inv_t::UART_SIGNAL_RXD_INV (C++ enumerator), 1384
uart_signal_inv_t::UART_SIGNAL_TXD_INV (C++ enumerator), 1385
uart_stop_bits_t (C++ enum), 1383
uart_stop_bits_t::UART_STOP_BITS_1 (C++ enumerator), 1383
uart_stop_bits_t::UART_STOP_BITS_1_5 (C++ enumerator), 1383
uart_stop_bits_t::UART_STOP_BITS_2 (C++ enumerator), 1383
uart_stop_bits_t::UART_STOP_BITS_MAX (C++ enumerator), 1383
uart_sw_flowctrl_t (C++ struct), 1381
uart_sw_flowctrl_t::xoff_char (C++ member), 1381
uart_sw_flowctrl_t::xoff_thrd (C++ member), 1382
uart_sw_flowctrl_t::xon_char (C++ member), 1381
uart_sw_flowctrl_t::xon_thrd (C++ member), 1381
uart_tx_chars (C++ function), 1373
UART_TXD_GPIO10_DIRECT_CHANNEL (C macro), 1386
UART_TXD_GPIO17_DIRECT_CHANNEL (C macro), 1387
UART_TXD_GPIO1_DIRECT_CHANNEL (C macro), 1386
uart_wait_tx_done (C++ function), 1373
uart_wait_tx_idle_polling (C++ function), 1378
uart_word_length_t (C++ enum), 1383
uart_word_length_t::UART_DATA_5_BITS (C++ enumerator), 1383
uart_word_length_t::UART_DATA_6_BITS (C++ enumerator), 1383
uart_word_length_t::UART_DATA_7_BITS (C++ enumerator), 1383
uart_word_length_t::UART_DATA_8_BITS (C++ enumerator), 1383
uart_word_length_t::UART_DATA_BITS_MAX (C++ enumerator), 1383
uart_write_bytes (C++ function), 1373
uart_write_bytes_with_break (C++ function), 1374
UINT16 (C++ type), 597
UINT32 (C++ type), 597
UINT64 (C++ type), 597
UINT8 (C++ type), 597
ulp_insn (C++ union), 2166
ulp_insn::adc (C++ member), 2170
ulp_insn::addr (C++ member), 2168
ulp_insn::alu_imm (C++ member), 2169
ulp_insn::alu_reg (C++ member), 2169
ulp_insn::alu_reg_s (C++ member), 2169
ulp_insn::b (C++ member), 2169
ulp_insn::bs (C++ member), 2169
ulp_insn::bx (C++ member), 2168
ulp_insn::cmp (C++ member), 2168
ulp_insn::cycle_sel (C++ member), 2170
ulp_insn::cycles (C++ member), 2166
ulp_insn::data (C++ member), 2169
ulp_insn::delay (C++ member), 2167
ulp_insn::dreg (C++ member), 2167
ulp_insn::end (C++ member), 2170
ulp_insn::halt (C++ member), 2168
ulp_insn::high (C++ member), 2169
ulp_insn::high_bits (C++ member), 2170
ulp_insn::i2c (C++ member), 2170
ulp_insn::i2c_addr (C++ member), 2170
ulp_insn::i2c_sel (C++ member), 2170
ulp_insn::imm (C++ member), 2168
ulp_insn::instruction (C++ member), 2171
ulp_insn::label (C++ member), 2170
ulp_insn::ld (C++ member), 2168
ulp_insn::low (C++ member), 2169
ulp_insn::low_bits (C++ member), 2170
ulp_insn::macro (C++ member), 2170
ulp_insn::mux (C++ member), 2169
ulp_insn::offset (C++ member), 2167
ulp_insn::opcode (C++ member), 2166
ulp_insn::periph_sel (C++ member), 2169
ulp_insn::rd_reg (C++ member), 2169
ulp_insn::reg (C++ member), 2168
ulp_insn::reserved (C++ member), 2170
ulp_insn::rw (C++ member), 2170
ulp_insn::sar_sel (C++ member), 2169
ulp_insn::sel (C++ member), 2169

ulp_insn::sign (*C++ member*), 2168
ulp_insn::sleep (*C++ member*), 2170
ulp_insn::sreg (*C++ member*), 2167
ulp_insn::st (*C++ member*), 2168
ulp_insn::sub_opcode (*C++ member*), 2167
ulp_insn::treg (*C++ member*), 2169
ulp_insn::tsens (*C++ member*), 2170
ulp_insn::type (*C++ member*), 2168
ulp_insn::unused (*C++ member*), 2166
ulp_insn::unused1 (*C++ member*), 2167
ulp_insn::unused2 (*C++ member*), 2167
ulp_insn::wait_delay (*C++ member*), 2170
ulp_insn::wakeup (*C++ member*), 2170
ulp_insn::wr_reg (*C++ member*), 2169
ulp_insn_t (*C++ type*), 2183
ulp_load_binary (*C++ function*), 2182
ulp_process_macros_and_load (*C++ function*), 2182
ulp_run (*C++ function*), 2182
ulp_set_wakeup_period (*C++ function*), 2183
ulp_timer_resume (*C++ function*), 2183
ulp_timer_stop (*C++ function*), 2183
ulTaskGenericNotifyTake (*C++ function*), 1927
ulTaskGenericNotifyValueClear (*C++ function*), 1929
ulTaskGetIdleRunTimeCounter (*C++ function*), 1922
ulTaskNotifyTake (*C macro*), 1933
ulTaskNotifyTakeIndexed (*C macro*), 1934
ulTaskNotifyValueClear (*C macro*), 1934
ulTaskNotifyValueClearIndexed (*C macro*), 1934
uxQueueMessagesWaiting (*C++ function*), 1939
uxQueueMessagesWaitingFromISR (*C++ function*), 1941
uxQueueSpacesAvailable (*C++ function*), 1939
uxSemaphoreGetCount (*C macro*), 1967
uxTaskGetNumberOfTasks (*C++ function*), 1917
uxTaskGetStackHighWaterMark (*C++ function*), 1917
uxTaskGetStackHighWaterMark2 (*C++ function*), 1917
uxTaskGetSystemState (*C++ function*), 1919
uxTaskPriorityGet (*C++ function*), 1910
uxTaskPriorityGetFromISR (*C++ function*), 1911
uxTimerGetReloadMode (*C++ function*), 1975

V

vApplicationGetIdleTaskMemory (*C++ function*), 1919
vApplicationGetTimerTaskMemory (*C++ function*), 1975
VENDOR_HCI_CMD_MASK (*C macro*), 300
vendor_ie_data_t (*C++ struct*), 903
vendor_ie_data_t::element_id (*C++ member*), 903
vendor_ie_data_t::length (*C++ member*), 903
vendor_ie_data_t::payload (*C++ member*), 903
vendor_ie_data_t::vendor_oui (*C++ member*), 903
vendor_ie_data_t::vendor_oui_type (*C++ member*), 903
vEventGroupDelete (*C++ function*), 1992
vMessageBufferDelete (*C macro*), 2009
vprintf_like_t (*C++ type*), 2080
vQueueAddToRegistry (*C++ function*), 1941
vQueueDelete (*C++ function*), 1939
vQueueUnregisterQueue (*C++ function*), 1942
vRingbufferDelete (*C++ function*), 2026
vRingbufferGetInfo (*C++ function*), 2027
vRingbufferReturnItem (*C++ function*), 2026
vRingbufferReturnItemFromISR (*C++ function*), 2026
vSemaphoreCreateBinary (*C macro*), 1954
vSemaphoreDelete (*C macro*), 1966
vStreamBufferDelete (*C++ function*), 1998
vTaskAllocateMPURegions (*C++ function*), 1906
vTaskDelay (*C++ function*), 1909
vTaskDelayUntil (*C macro*), 1932
vTaskDelete (*C++ function*), 1908
vTaskEndScheduler (*C++ function*), 1915
vTaskGenericNotifyGiveFromISR (*C++ function*), 1926
vTaskGetInfo (*C++ function*), 1911
vTaskGetRunTimeStats (*C++ function*), 1922
vTaskList (*C++ function*), 1921
vTaskNotifyGiveFromISR (*C macro*), 1933
vTaskNotifyGiveIndexedFromISR (*C macro*), 1933
vTaskPrioritySet (*C++ function*), 1912
vTaskResume (*C++ function*), 1913
vTaskSetApplicationTaskTag (*C++ function*), 1918
vTaskSetThreadLocalStoragePointer (*C++ function*), 1918
vTaskSetThreadLocalStoragePointerAndDelCallback (*C++ function*), 1919
vTaskSetTimeOutState (*C++ function*), 1929
vTaskStartScheduler (*C++ function*), 1914
vTaskSuspend (*C++ function*), 1913
vTaskSuspendAll (*C++ function*), 1915
vTimerSetReloadMode (*C++ function*), 1974
vTimerSetTimerID (*C++ function*), 1972

W

wifi_action_rx_cb_t (*C++ type*), 916
wifi_action_tx_req_t (*C++ struct*), 908
wifi_action_tx_req_t::data (*C++ member*), 908
wifi_action_tx_req_t::data_len (*C++ member*), 908

wifi_action_tx_req_t::dest_mac (C++ member), 908
wifi_action_tx_req_t::ifx (C++ member), 908
wifi_action_tx_req_t::no_ack (C++ member), 908
wifi_action_tx_req_t::rx_cb (C++ member), 908
wifi_active_scan_time_t (C++ struct), 897
wifi_active_scan_time_t::max (C++ member), 897
wifi_active_scan_time_t::min (C++ member), 897
WIFI_AMPDU_RX_ENABLED (C macro), 895
WIFI_AMPDU_TX_ENABLED (C macro), 895
WIFI_AMSDU_TX_ENABLED (C macro), 895
wifi_ant_config_t (C++ struct), 907
wifi_ant_config_t::enabled_ant0 (C++ member), 908
wifi_ant_config_t::enabled_ant1 (C++ member), 908
wifi_ant_config_t::rx_ant_default (C++ member), 907
wifi_ant_config_t::rx_ant_mode (C++ member), 907
wifi_ant_config_t::tx_ant_mode (C++ member), 907
wifi_ant_gpio_config_t (C++ struct), 907
wifi_ant_gpio_config_t::gpio_cfg (C++ member), 907
wifi_ant_gpio_t (C++ struct), 907
wifi_ant_gpio_t::gpio_num (C++ member), 907
wifi_ant_gpio_t::gpio_select (C++ member), 907
wifi_ant_mode_t (C++ enum), 924
wifi_ant_mode_t::WIFI_ANT_MODE_ANT0 (C++ enumerator), 924
wifi_ant_mode_t::WIFI_ANT_MODE_ANT1 (C++ enumerator), 925
wifi_ant_mode_t::WIFI_ANT_MODE_AUTO (C++ enumerator), 925
wifi_ant_mode_t::WIFI_ANT_MODE_MAX (C++ enumerator), 925
wifi_ant_t (C++ enum), 922
wifi_ant_t::WIFI_ANT_ANT0 (C++ enumerator), 922
wifi_ant_t::WIFI_ANT_ANT1 (C++ enumerator), 922
wifi_ant_t::WIFI_ANT_MAX (C++ enumerator), 922
wifi_ap_config_t (C++ struct), 900
wifi_ap_config_t::authmode (C++ member), 900
wifi_ap_config_t::beacon_interval (C++ member), 900
wifi_ap_config_t::channel (C++ member), 900
wifi_ap_config_t::ftm_responder (C++ member), 901
wifi_ap_config_t::max_connection (C++ member), 900
wifi_ap_config_t::pairwise_cipher (C++ member), 900
wifi_ap_config_t::password (C++ member), 900
wifi_ap_config_t::pmf_cfg (C++ member), 901
wifi_ap_config_t::ssid (C++ member), 900
wifi_ap_config_t::ssid_hidden (C++ member), 900
wifi_ap_config_t::ssid_len (C++ member), 900
wifi_ap_record_t (C++ struct), 898
wifi_ap_record_t::ant (C++ member), 899
wifi_ap_record_t::authmode (C++ member), 898
wifi_ap_record_t::bssid (C++ member), 898
wifi_ap_record_t::country (C++ member), 899
wifi_ap_record_t::ftm_initiator (C++ member), 899
wifi_ap_record_t::ftm_responder (C++ member), 899
wifi_ap_record_t::group_cipher (C++ member), 899
wifi_ap_record_t::pairwise_cipher (C++ member), 898
wifi_ap_record_t::phy_11b (C++ member), 899
wifi_ap_record_t::phy_11g (C++ member), 899
wifi_ap_record_t::phy_11n (C++ member), 899
wifi_ap_record_t::phy_lr (C++ member), 899
wifi_ap_record_t::primary (C++ member), 898
wifi_ap_record_t::reserved (C++ member), 899
wifi_ap_record_t::rssi (C++ member), 898
wifi_ap_record_t::second (C++ member), 898
wifi_ap_record_t::ssid (C++ member), 898
wifi_ap_record_t::wps (C++ member), 899
wifi_auth_mode_t (C++ enum), 917
wifi_auth_mode_t::WIFI_AUTH_ENTERPRISE (C++ enumerator), 917
wifi_auth_mode_t::WIFI_AUTH_MAX (C++ enumerator), 918
wifi_auth_mode_t::WIFI_AUTH_OPEN (C++ enumerator), 917
wifi_auth_mode_t::WIFI_AUTH_OWE (C++ enumerator), 918
wifi_auth_mode_t::WIFI_AUTH_WAPI_PSK (C++ enumerator), 918

wifi_auth_mode_t::WIFI_AUTH_WEP (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA2_ENTERPRISE (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA2_PSK (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA2_WPA3_PSK (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA3_ENT_192 (*C++ enumerator*), 918
wifi_auth_mode_t::WIFI_AUTH_WPA3_PSK (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA_PSK (*C++ enumerator*), 917
wifi_auth_mode_t::WIFI_AUTH_WPA_WPA2_PSK (*C++ enumerator*), 917
wifi_bandwidth_t (*C++ enum*), 922
wifi_bandwidth_t::WIFI_BW_HT20 (*C++ enumerator*), 923
wifi_bandwidth_t::WIFI_BW_HT40 (*C++ enumerator*), 923
WIFI_CACHE_TX_BUFFER_NUM (*C macro*), 894
wifi_cipher_type_t (*C++ enum*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_AES_CMAC128 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_AES_GMAC128 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_AES_GMAC256 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_CCMP (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_GCMP (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_GCMP256 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_NONE (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_SMS4 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_TKIP (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_TKIP_CCMP (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_UNKNOWN (*C++ enumerator*), 922
wifi_cipher_type_t::WIFI_CIPHER_TYPE_WEP104 (*C++ enumerator*), 921
wifi_cipher_type_t::WIFI_CIPHER_TYPE_WEP40 (*C++ enumerator*), 921
wifi_config_t (*C++ union*), 896
wifi_config_t::ap (*C++ member*), 896
wifi_config_t::sta (*C++ member*), 896
wifi_country_policy_t (*C++ enum*), 917
wifi_country_policy_t::WIFI_COUNTRY_POLICY_AUTO (*C++ enumerator*), 917
wifi_country_policy_t::WIFI_COUNTRY_POLICY_MANUAL (*C++ enumerator*), 917
wifi_country_t (*C++ struct*), 897
wifi_country_t::cc (*C++ member*), 897
wifi_country_t::max_tx_power (*C++ member*), 897
wifi_country_t::nchan (*C++ member*), 897
wifi_country_t::policy (*C++ member*), 897
wifi_country_t::schan (*C++ member*), 897
wifi_csi_cb_t (*C++ type*), 896
wifi_csi_config_t (*C++ struct*), 906
wifi_csi_config_t::channel_filter_en (*C++ member*), 906
wifi_csi_config_t::dump_ack_en (*C++ member*), 906
wifi_csi_config_t::htltf_en (*C++ member*), 906
wifi_csi_config_t::lltf_en (*C++ member*), 906
wifi_csi_config_t::ltf_merge_en (*C++ member*), 906
wifi_csi_config_t::manu_scale (*C++ member*), 906
wifi_csi_config_t::shift (*C++ member*), 906
wifi_csi_config_t::stbc_htltf2_en (*C++ member*), 906
WIFI_CSI_ENABLED (*C macro*), 895
wifi_csi_info_t (*C++ struct*), 906
wifi_csi_info_t::buf (*C++ member*), 907
wifi_csi_info_t::dmac (*C++ member*), 907
wifi_csi_info_t::first_word_invalid (*C++ member*), 907
wifi_csi_info_t::len (*C++ member*), 907
wifi_csi_info_t::mac (*C++ member*), 906
wifi_csi_info_t::rx_ctrl (*C++ member*), 906
WIFI_DEFAULT_RX_BA_WIN (*C macro*), 895
WIFI_DYNAMIC_TX_BUFFER_NUM (*C macro*), 894
WIFI_ENABLE_SPIRAM (*C macro*), 895
WIFI_ENABLE_WPA3_SAE (*C macro*), 895
wifi_err_reason_t (*C++ enum*), 918
wifi_err_reason_t::WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT (*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_802_1X_AUTH_FAILED (*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_AKMP_INVALID (*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_ALTERATIVE_CHANNEL_OCCUPIED (*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_AP_INITIATED (*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_AP_TSF_RESET (*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_ASSOC_COMEBACK_TIME_TOO_LONG (*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_ASSOC_EXPIRE (*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_ASSOC_FAIL (*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_ASSOC_LEAVE

(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_ASSOC_NOT_AUTHED
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_ASSOC_TOOMANY
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_AUTH_EXPIRE
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_AUTH_FAIL
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_AUTH_LEAVE
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_BAD_CIPHER_OR_AKM
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_BEACON_TIMEOUT
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_BSS_TRANSITION_DISASSOC
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_CIPHER_SUITE_REJECTED
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_CONNECTION_FAIL
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_DISASSOC_PWRCAP_BAD
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_DISASSOC_SUPCHAN_BAD
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_END_BA
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_EXCEEDED_TXOP
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_GROUP_CIPHER_INVALID
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_HANDSHAKE_TIMEOUT
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_IE_IN_4WAY_DIFFERS
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_IE_INVALID
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_INVALID_FTIE_CONTENT
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_INVALID_MDE
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_INVALID_PMKID
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_INVALID_RSN_IE_CAP
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_MIC_FAILURE
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_MISSING_ACKS
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_NO_AP_FOUND
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_NO_SSP_ROAMING_AGREEMENT
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_NOT_ASSOCED
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_NOT_AUTHED
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_NOT_AUTHORIZED_THIS
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_NOT_ENOUGH_BANDWIDTH
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_PAIRWISE_CIPHER_INVALID
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_PEER_INITIATED
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_ROAMING
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_SA_QUERY_TIMEOUT
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_SERVICE_CHANGE_PERCLUDES
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_SSP_REQUESTED_DISASSOC
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_STA_LEAVING
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_TDLS_PEER_UNREACHABLE
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_TDLS_UNSPECIFIED
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_TIMEOUT
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_TRANSMISSION_LINK_
(*C++ enumerator*), 920
wifi_err_reason_t::WIFI_REASON_UNKNOWN_BA
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_UNSPECIFIED
(*C++ enumerator*), 918
wifi_err_reason_t::WIFI_REASON_UNSPECIFIED_QOS
(*C++ enumerator*), 919
wifi_err_reason_t::WIFI_REASON_UNSUPP_RSN_IE_VERSION
(*C++ enumerator*), 919
wifi_event_action_tx_status_t (*C++ struct*), 913
wifi_event_action_tx_status_t::context
(*C++ member*), 913
wifi_event_action_tx_status_t::da
(*C++ member*), 913
wifi_event_action_tx_status_t::ifx
(*C++ member*), 913
wifi_event_action_tx_status_t::status
(*C++ member*), 913
wifi_event_ap_probe_req_rx_t (*C++ struct*), 911
wifi_event_ap_probe_req_rx_t::mac
(*C++ member*), 911
wifi_event_ap_probe_req_rx_t::rssi
(*C++ member*), 911
wifi_event_ap_staconnected_t (*C++ struct*), 911
wifi_event_ap_staconnected_t::aid
(*C++ member*), 911
wifi_event_ap_staconnected_t::is_mesh_child

(C++ member), 911
wifi_event_ap_staconnected_t::mac (C++ member), 911
wifi_event_ap_stadisconnected_t (C++ struct), 911
wifi_event_ap_stadisconnected_t::aid (C++ member), 911
wifi_event_ap_stadisconnected_t::is_mesh_child (C++ member), 911
wifi_event_ap_stadisconnected_t::mac (C++ member), 911
wifi_event_ap_wps_rg_fail_reason_t (C++ struct), 913
wifi_event_ap_wps_rg_fail_reason_t::peer_macaddr (C++ member), 914
wifi_event_ap_wps_rg_fail_reason_t::reason (C++ member), 914
wifi_event_ap_wps_rg_pin_t (C++ struct), 913
wifi_event_ap_wps_rg_pin_t::pin_code (C++ member), 913
wifi_event_ap_wps_rg_success_t (C++ struct), 914
wifi_event_ap_wps_rg_success_t::peer_macaddr (C++ member), 914
wifi_event_bss_rssi_low_t (C++ struct), 911
wifi_event_bss_rssi_low_t::rssi (C++ member), 912
wifi_event_ftm_report_t (C++ struct), 912
wifi_event_ftm_report_t::dist_est (C++ member), 912
wifi_event_ftm_report_t::ftm_report_data (C++ member), 913
wifi_event_ftm_report_t::ftm_report_num_entries (C++ member), 913
wifi_event_ftm_report_t::peer_mac (C++ member), 912
wifi_event_ftm_report_t::rtt_est (C++ member), 912
wifi_event_ftm_report_t::rtt_raw (C++ member), 912
wifi_event_ftm_report_t::status (C++ member), 912
WIFI_EVENT_MASK_ALL (C macro), 915
WIFI_EVENT_MASK_AP_PROBEREQRECVED (C macro), 916
WIFI_EVENT_MASK_NONE (C macro), 915
wifi_event_roc_done_t (C++ struct), 913
wifi_event_roc_done_t::context (C++ member), 913
wifi_event_sta_authmode_change_t (C++ struct), 910
wifi_event_sta_authmode_change_t::new_mode (C++ member), 910
wifi_event_sta_authmode_change_t::old_mode (C++ member), 910
wifi_event_sta_connected_t (C++ struct), 909
wifi_event_sta_connected_t::authmode (C++ member), 909
wifi_event_sta_connected_t::bssid (C++ member), 909
wifi_event_sta_connected_t::channel (C++ member), 909
wifi_event_sta_connected_t::ssid (C++ member), 909
wifi_event_sta_connected_t::ssid_len (C++ member), 909
wifi_event_sta_disconnected_t (C++ struct), 909
wifi_event_sta_disconnected_t::bssid (C++ member), 910
wifi_event_sta_disconnected_t::reason (C++ member), 910
wifi_event_sta_disconnected_t::rssi (C++ member), 910
wifi_event_sta_disconnected_t::ssid (C++ member), 909
wifi_event_sta_disconnected_t::ssid_len (C++ member), 910
wifi_event_sta_scan_done_t (C++ struct), 909
wifi_event_sta_scan_done_t::number (C++ member), 909
wifi_event_sta_scan_done_t::scan_id (C++ member), 909
wifi_event_sta_scan_done_t::status (C++ member), 909
wifi_event_sta_wps_er_pin_t (C++ struct), 910
wifi_event_sta_wps_er_pin_t::pin_code (C++ member), 910
wifi_event_sta_wps_er_success_t (C++ struct), 910
wifi_event_sta_wps_er_success_t::ap_cred (C++ member), 911
wifi_event_sta_wps_er_success_t::ap_cred_cnt (C++ member), 910
wifi_event_sta_wps_er_success_t::passphrase (C++ member), 910
wifi_event_sta_wps_er_success_t::ssid (C++ member), 910
wifi_event_sta_wps_fail_reason_t (C++ enum), 929
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON (C++ enumerator), 929
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON (C++ enumerator), 929
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON (C++ enumerator), 929
wifi_event_t (C++ enum), 927
wifi_event_t::WIFI_EVENT_ACTION_TX_STATUS (C++ enumerator), 928
wifi_event_t::WIFI_EVENT_AP_PROBEREQRECVED (C++ enumerator), 928
wifi_event_t::WIFI_EVENT_AP_STACONNECTED

(C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_STADISCONNECTED (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_START (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_STOP (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_WPS_RG_FAILED (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_WPS_RG_PBC_OVERLAP (C++ *enumerator*), 929
wifi_event_t::WIFI_EVENT_AP_WPS_RG_PIN (C++ *enumerator*), 929
wifi_event_t::WIFI_EVENT_AP_WPS_RG_SUCCESS (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_AP_WPS_RG_TIMEOUT (C++ *enumerator*), 929
wifi_event_t::WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_FTM_REPORT (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_MAX (C++ *enumerator*), 929
wifi_event_t::WIFI_EVENT_ROC_DONE (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_SCAN_DONE (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_AUTHMODE_CHANGE (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_BEACON_TIMEOUT (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_STA_CONNECTED (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_DISCONNECTED (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_START (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_STOP (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_WPS_ER_FAILED (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_WPS_ER_PBC_OVERLAP (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_STA_WPS_ER_PIN (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_STA_WPS_ER_SUCCESS (C++ *enumerator*), 927
wifi_event_t::WIFI_EVENT_STA_WPS_ER_TIMEOUT (C++ *enumerator*), 928
wifi_event_t::WIFI_EVENT_WIFI_READY (C++ *enumerator*), 927
WIFI_FEATURE_CAPS (*C macro*), 895
WIFI_FTM_INITIATOR (*C macro*), 895
wifi_ftm_initiator_cfg_t (*C++ struct*), 908
wifi_ftm_initiator_cfg_t::burst_period (C++ *member*), 908
wifi_ftm_initiator_cfg_t::channel (C++ *member*), 908
wifi_ftm_initiator_cfg_t::frm_count (C++ *member*), 908
wifi_ftm_initiator_cfg_t::resp_mac (C++ *member*), 908
wifi_ftm_initiator_cfg_t::use_get_report_api (C++ *member*), 909
wifi_ftm_report_entry_t (*C++ struct*), 912
wifi_ftm_report_entry_t::dlog_token (C++ *member*), 912
wifi_ftm_report_entry_t::rssi (C++ *member*), 912
wifi_ftm_report_entry_t::rtt (C++ *member*), 912
wifi_ftm_report_entry_t::t1 (C++ *member*), 912
wifi_ftm_report_entry_t::t2 (C++ *member*), 912
wifi_ftm_report_entry_t::t3 (C++ *member*), 912
wifi_ftm_report_entry_t::t4 (C++ *member*), 912
WIFI_FTM_RESPONDER (*C macro*), 895
wifi_ftm_status_t (*C++ enum*), 929
wifi_ftm_status_t::FTM_STATUS_CONF_REJECTED (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_FAIL (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_NO_RESPONSE (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_NO_VALID_MSMT (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_SUCCESS (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_UNSUPPORTED (C++ *enumerator*), 929
wifi_ftm_status_t::FTM_STATUS_USER_TERM (C++ *enumerator*), 929
WIFI_INIT_CONFIG_DEFAULT (*C macro*), 895
WIFI_INIT_CONFIG_MAGIC (*C macro*), 895
wifi_init_config_t (*C++ struct*), 891
wifi_init_config_t::ampdu_rx_enable (C++ *member*), 892
wifi_init_config_t::ampdu_tx_enable (C++ *member*), 892
wifi_init_config_t::amsdu_tx_enable (C++ *member*), 892
wifi_init_config_t::beacon_max_len (C++ *member*), 893
wifi_init_config_t::cache_tx_buf_num (C++ *member*), 892
wifi_init_config_t::csi_enable (C++ *member*), 892
wifi_init_config_t::dynamic_rx_buf_num (C++ *member*), 892
wifi_init_config_t::dynamic_tx_buf_num (C++ *member*), 892

wifi_init_config_t::espnow_max_encrypt_num (C++ member), 893
wifi_init_config_t::feature_caps (C++ member), 893
wifi_init_config_t::magic (C++ member), 893
wifi_init_config_t::mgmt_sbuf_num (C++ member), 893
wifi_init_config_t::nano_enable (C++ member), 892
wifi_init_config_t::nvs_enable (C++ member), 892
wifi_init_config_t::osi_funcs (C++ member), 891
wifi_init_config_t::rx_ba_win (C++ member), 892
wifi_init_config_t::rx_mgmt_buf_num (C++ member), 892
wifi_init_config_t::rx_mgmt_buf_type (C++ member), 892
wifi_init_config_t::sta_disconnected_pm (C++ member), 893
wifi_init_config_t::static_rx_buf_num (C++ member), 892
wifi_init_config_t::static_tx_buf_num (C++ member), 892
wifi_init_config_t::tx_buf_type (C++ member), 892
wifi_init_config_t::wifi_task_core_id (C++ member), 893
wifi_init_config_t::wpa_crypto_funcs (C++ member), 892
wifi_interface_t (C++ enum), 917
wifi_interface_t::WIFI_IF_AP (C++ enumerator), 917
wifi_interface_t::WIFI_IF_STA (C++ enumerator), 917
WIFI_MGMT_SBUF_NUM (C macro), 895
wifi_mode_t (C++ enum), 916
wifi_mode_t::WIFI_MODE_AP (C++ enumerator), 916
wifi_mode_t::WIFI_MODE_APSTA (C++ enumerator), 916
wifi_mode_t::WIFI_MODE_MAX (C++ enumerator), 917
wifi_mode_t::WIFI_MODE_NULL (C++ enumerator), 916
wifi_mode_t::WIFI_MODE_STA (C++ enumerator), 916
WIFI_NANO_FORMAT_ENABLED (C macro), 895
WIFI_NVS_ENABLED (C macro), 895
WIFI_OFFCHAN_TX_CANCEL (C macro), 914
WIFI_OFFCHAN_TX_REQ (C macro), 914
wifi_phy_mode_t (C++ enum), 924
wifi_phy_mode_t::WIFI_PHY_MODE_11B (C++ enumerator), 924
wifi_phy_mode_t::WIFI_PHY_MODE_11G (C++ enumerator), 924
wifi_phy_mode_t::WIFI_PHY_MODE_HE20 (C++ enumerator), 924
wifi_phy_mode_t::WIFI_PHY_MODE_HT20 (C++ enumerator), 924
wifi_phy_mode_t::WIFI_PHY_MODE_HT40 (C++ enumerator), 924
wifi_phy_mode_t::WIFI_PHY_MODE_LR (C++ enumerator), 924
wifi_phy_rate_t (C++ enum), 925
wifi_phy_rate_t::WIFI_PHY_RATE_11M_L (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_11M_S (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_12M (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_18M (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_1M_L (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_24M (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_2M_L (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_2M_S (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_36M (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_48M (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_54M (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_5M_L (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_5M_S (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_6M (C++ enumerator), 925
wifi_phy_rate_t::WIFI_PHY_RATE_9M (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_250K (C++ enumerator), 927
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_500K (C++ enumerator), 927
wifi_phy_rate_t::WIFI_PHY_RATE_MAX (C++ enumerator), 927
wifi_phy_rate_t::WIFI_PHY_RATE_MCS0_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS0_SGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS1_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS1_SGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS2_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS2_SGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS3_LGI

(C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS3_SGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS4_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS4_SGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS5_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS5_SGI (C++ enumerator), 927
wifi_phy_rate_t::WIFI_PHY_RATE_MCS6_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS6_SGI (C++ enumerator), 927
wifi_phy_rate_t::WIFI_PHY_RATE_MCS7_LGI (C++ enumerator), 926
wifi_phy_rate_t::WIFI_PHY_RATE_MCS7_SGI (C++ enumerator), 927
wifi_pkt_rx_ctrl_t (C++ struct), 903
wifi_pkt_rx_ctrl_t::__pad0__ (C++ member), 904
wifi_pkt_rx_ctrl_t::__pad1__ (C++ member), 904
wifi_pkt_rx_ctrl_t::__pad2__ (C++ member), 904
wifi_pkt_rx_ctrl_t::__pad3__ (C++ member), 904
wifi_pkt_rx_ctrl_t::__pad4__ (C++ member), 905
wifi_pkt_rx_ctrl_t::__pad5__ (C++ member), 905
wifi_pkt_rx_ctrl_t::__pad6__ (C++ member), 905
wifi_pkt_rx_ctrl_t::__pad7__ (C++ member), 905
wifi_pkt_rx_ctrl_t::aggregation (C++ member), 904
wifi_pkt_rx_ctrl_t::ampdu_cnt (C++ member), 905
wifi_pkt_rx_ctrl_t::ant (C++ member), 905
wifi_pkt_rx_ctrl_t::channel (C++ member), 905
wifi_pkt_rx_ctrl_t::cwb (C++ member), 904
wifi_pkt_rx_ctrl_t::fec_coding (C++ member), 904
wifi_pkt_rx_ctrl_t::mcs (C++ member), 904
wifi_pkt_rx_ctrl_t::noise_floor (C++ member), 904
wifi_pkt_rx_ctrl_t::not_sounding (C++ member), 904
wifi_pkt_rx_ctrl_t::rate (C++ member), 904
wifi_pkt_rx_ctrl_t::rssi (C++ member), 904
wifi_pkt_rx_ctrl_t::rx_state (C++ member), 905
wifi_pkt_rx_ctrl_t::secondary_channel (C++ member), 905
wifi_pkt_rx_ctrl_t::sgi (C++ member), 904
wifi_pkt_rx_ctrl_t::sig_len (C++ member), 905
wifi_pkt_rx_ctrl_t::sig_mode (C++ member), 904
wifi_pkt_rx_ctrl_t::smoothing (C++ member), 904
wifi_pkt_rx_ctrl_t::stbc (C++ member), 904
wifi_pkt_rx_ctrl_t::timestamp (C++ member), 905
wifi_pmf_config_t (C++ struct), 899
wifi_pmf_config_t::capable (C++ member), 900
wifi_pmf_config_t::required (C++ member), 900
WIFI_PROMIS_CTRL_FILTER_MASK_ACK (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_ALL (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_BA (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_BAR (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_CFEND (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_CTS (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_RTS (C macro), 915
WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER (C macro), 915
WIFI_PROMIS_FILTER_MASK_ALL (C macro), 914
WIFI_PROMIS_FILTER_MASK_CTRL (C macro), 914
WIFI_PROMIS_FILTER_MASK_DATA (C macro), 914
WIFI_PROMIS_FILTER_MASK_DATA_AMPDU (C macro), 915
WIFI_PROMIS_FILTER_MASK_DATA_MPDU (C macro), 915
WIFI_PROMIS_FILTER_MASK_FCSFAIL (C macro), 915
WIFI_PROMIS_FILTER_MASK_MGMT (C macro), 914
WIFI_PROMIS_FILTER_MASK_MISC (C macro), 915
wifi_promiscuous_cb_t (C++ type), 896
wifi_promiscuous_filter_t (C++ struct), 905
wifi_promiscuous_filter_t::filter_mask (C++ member), 906
wifi_promiscuous_pkt_t (C++ struct), 905

wifi_promiscuous_pkt_t::payload (C++ member), 905
wifi_promiscuous_pkt_t::rx_ctrl (C++ member), 905
wifi_promiscuous_pkt_type_t (C++ enum), 924
wifi_promiscuous_pkt_type_t::WIFI_PKT_CTRL (C++ enumerator), 924
wifi_promiscuous_pkt_type_t::WIFI_PKT_DATA (C++ enumerator), 924
wifi_promiscuous_pkt_type_t::WIFI_PKT_MGMT (C++ enumerator), 924
wifi_promiscuous_pkt_type_t::WIFI_PKT_MISC (C++ enumerator), 924
WIFI_PROTOCOL_11B (C macro), 914
WIFI_PROTOCOL_11G (C macro), 914
WIFI_PROTOCOL_11N (C macro), 914
WIFI_PROTOCOL_LR (C macro), 914
wifi_prov_cb_event_t (C++ enum), 1709
wifi_prov_cb_event_t::WIFI_PROV_CRED_FAIL (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_CRED_RECV (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_CRED_SUCCESS (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_DEINIT (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_END (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_INIT (C++ enumerator), 1709
wifi_prov_cb_event_t::WIFI_PROV_START (C++ enumerator), 1709
wifi_prov_cb_func_t (C++ type), 1709
wifi_prov_config_data_handler (C++ function), 1712
wifi_prov_config_get_data_t (C++ struct), 1712
wifi_prov_config_get_data_t::conn_info (C++ member), 1712
wifi_prov_config_get_data_t::fail_reason (C++ member), 1712
wifi_prov_config_get_data_t::wifi_state (C++ member), 1712
wifi_prov_config_handlers (C++ struct), 1713
wifi_prov_config_handlers::apply_config_handler (C++ member), 1713
wifi_prov_config_handlers::ctx (C++ member), 1713
wifi_prov_config_handlers::get_status_handler (C++ member), 1713
wifi_prov_config_handlers::set_config_handler (C++ member), 1713
wifi_prov_config_handlers_t (C++ type), 1713
wifi_prov_config_set_data_t (C++ struct), 1712
wifi_prov_config_set_data_t::bssid (C++ member), 1713
wifi_prov_config_set_data_t::channel (C++ member), 1713
wifi_prov_config_set_data_t::password (C++ member), 1713
wifi_prov_config_set_data_t::ssid (C++ member), 1713
wifi_prov_ctx_t (C++ type), 1713
WIFI_PROV_EVENT_HANDLER_NONE (C macro), 1708
wifi_prov_event_handler_t (C++ struct), 1707
wifi_prov_event_handler_t::event_cb (C++ member), 1707
wifi_prov_event_handler_t::user_data (C++ member), 1707
wifi_prov_mgr_config_t (C++ struct), 1708
wifi_prov_mgr_config_t::app_event_handler (C++ member), 1708
wifi_prov_mgr_config_t::scheme (C++ member), 1708
wifi_prov_mgr_config_t::scheme_event_handler (C++ member), 1708
wifi_prov_mgr_configure_sta (C++ function), 1706
wifi_prov_mgr_deinit (C++ function), 1702
wifi_prov_mgr_disable_auto_stop (C++ function), 1704
wifi_prov_mgr_endpoint_create (C++ function), 1705
wifi_prov_mgr_endpoint_register (C++ function), 1705
wifi_prov_mgr_endpoint_unregister (C++ function), 1706
wifi_prov_mgr_get_wifi_disconnect_reason (C++ function), 1706
wifi_prov_mgr_get_wifi_state (C++ function), 1706
wifi_prov_mgr_init (C++ function), 1702
wifi_prov_mgr_is_provisioned (C++ function), 1702
wifi_prov_mgr_reset_provisioning (C++ function), 1707
wifi_prov_mgr_reset_sm_state_on_failure (C++ function), 1707
wifi_prov_mgr_set_app_info (C++ function), 1704
wifi_prov_mgr_start_provisioning (C++ function), 1703
wifi_prov_mgr_stop_provisioning (C++ function), 1704
wifi_prov_mgr_wait (C++ function), 1704
wifi_prov_scheme (C++ struct), 1707
wifi_prov_scheme::delete_config (C++ member), 1708
wifi_prov_scheme::new_config (C++ member), 1707

wifi_prov_scheme::prov_start (*C++ member*), 1707
wifi_prov_scheme::prov_stop (*C++ member*), 1707
wifi_prov_scheme::set_config_endpoint (*C++ member*), 1708
wifi_prov_scheme::set_config_service (*C++ member*), 1708
wifi_prov_scheme::wifi_mode (*C++ member*), 1708
wifi_prov_scheme_ble_event_cb_free_ble (*C++ function*), 1710
wifi_prov_scheme_ble_event_cb_free_bt (*C++ function*), 1710
wifi_prov_scheme_ble_event_cb_free_btdm (*C++ function*), 1710
WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE (*C macro*), 1711
WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT (*C macro*), 1711
WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM (*C macro*), 1711
wifi_prov_scheme_ble_set_mfg_data (*C++ function*), 1710
wifi_prov_scheme_ble_set_service_uuid (*C++ function*), 1710
wifi_prov_scheme_softap_set_httpd_handle (*C++ function*), 1711
wifi_prov_scheme_t (*C++ type*), 1709
wifi_prov_security (*C++ enum*), 1710
wifi_prov_security2_params_t (*C++ type*), 1709
wifi_prov_security::WIFI_PROV_SECURITY_0 (*C++ enumerator*), 1710
wifi_prov_security::WIFI_PROV_SECURITY_1 (*C++ enumerator*), 1710
wifi_prov_security::WIFI_PROV_SECURITY_2 (*C++ enumerator*), 1710
wifi_prov_security_t (*C++ type*), 1709
wifi_prov_sta_conn_info_t (*C++ struct*), 1712
wifi_prov_sta_conn_info_t::auth_mode (*C++ member*), 1712
wifi_prov_sta_conn_info_t::bssid (*C++ member*), 1712
wifi_prov_sta_conn_info_t::channel (*C++ member*), 1712
wifi_prov_sta_conn_info_t::ip_addr (*C++ member*), 1712
wifi_prov_sta_conn_info_t::ssid (*C++ member*), 1712
wifi_prov_sta_fail_reason_t (*C++ enum*), 1714
wifi_prov_sta_fail_reason_t::WIFI_PROV_STA_AP_NOT_FOUND (*C++ enumerator*), 1714
wifi_prov_sta_fail_reason_t::WIFI_PROV_STA_AUTH_ERROR (*C++ enumerator*), 1714
wifi_prov_sta_state_t (*C++ enum*), 1714
wifi_prov_sta_state_t::WIFI_PROV_STA_CONNECTED (*C++ enumerator*), 1714
wifi_prov_sta_state_t::WIFI_PROV_STA_CONNECTING (*C++ enumerator*), 1714
wifi_prov_sta_state_t::WIFI_PROV_STA_DISCONNECTED (*C++ enumerator*), 1714
wifi_ps_type_t (*C++ enum*), 922
wifi_ps_type_t::WIFI_PS_MAX_MODEM (*C++ enumerator*), 922
wifi_ps_type_t::WIFI_PS_MIN_MODEM (*C++ enumerator*), 922
wifi_ps_type_t::WIFI_PS_NONE (*C++ enumerator*), 922
WIFI_ROC_CANCEL (*C macro*), 914
WIFI_ROC_REQ (*C macro*), 914
WIFI_RX_MGMT_BUF_NUM_DEF (*C macro*), 894
wifi_sae_pwe_method_t (*C++ enum*), 923
wifi_sae_pwe_method_t::WPA3_SAE_PWE_BOTH (*C++ enumerator*), 923
wifi_sae_pwe_method_t::WPA3_SAE_PWE_HASH_TO_ELEME (*C++ enumerator*), 923
wifi_sae_pwe_method_t::WPA3_SAE_PWE_HUNT_AND_PECK (*C++ enumerator*), 923
wifi_sae_pwe_method_t::WPA3_SAE_PWE_UNSPECIFIED (*C++ enumerator*), 923
wifi_scan_config_t (*C++ struct*), 897
wifi_scan_config_t::bssid (*C++ member*), 898
wifi_scan_config_t::channel (*C++ member*), 898
wifi_scan_config_t::home_chan_dwell_time (*C++ member*), 898
wifi_scan_config_t::scan_time (*C++ member*), 898
wifi_scan_config_t::scan_type (*C++ member*), 898
wifi_scan_config_t::show_hidden (*C++ member*), 898
wifi_scan_config_t::ssid (*C++ member*), 898
wifi_scan_method_t (*C++ enum*), 922
wifi_scan_method_t::WIFI_ALL_CHANNEL_SCAN (*C++ enumerator*), 922
wifi_scan_method_t::WIFI_FAST_SCAN (*C++ enumerator*), 922
wifi_scan_threshold_t (*C++ struct*), 899
wifi_scan_threshold_t::authmode (*C++ member*), 899
wifi_scan_threshold_t::rssi (*C++ member*), 899
wifi_scan_time_t (*C++ struct*), 897
wifi_scan_time_t::active (*C++ member*), 897
wifi_scan_time_t::passive (*C++ member*), 897
wifi_scan_type_t (*C++ enum*), 921
wifi_scan_type_t::WIFI_SCAN_TYPE_ACTIVE (*C++ enumerator*), 921

wifi_scan_type_t::WIFI_SCAN_TYPE_PASSIVE (C++ enumerator), 921
wifi_second_chan_t (C++ enum), 920
wifi_second_chan_t::WIFI_SECOND_CHAN_ABOVE (C++ enumerator), 920
wifi_second_chan_t::WIFI_SECOND_CHAN_BELOW (C++ enumerator), 920
wifi_second_chan_t::WIFI_SECOND_CHAN_NONE (C++ enumerator), 920
WIFI_SOFTAP_BEACON_MAX_LEN (C macro), 895
wifi_sort_method_t (C++ enum), 922
wifi_sort_method_t::WIFI_CONNECT_AP_BY_SECURITY (C++ enumerator), 922
wifi_sort_method_t::WIFI_CONNECT_AP_BY_SIGNAL (C++ enumerator), 922
wifi_sta_config_t (C++ struct), 901
wifi_sta_config_t::bssid (C++ member), 901
wifi_sta_config_t::bssid_set (C++ member), 901
wifi_sta_config_t::btm_enabled (C++ member), 902
wifi_sta_config_t::channel (C++ member), 901
wifi_sta_config_t::failure_retry_cnt (C++ member), 902
wifi_sta_config_t::ft_enabled (C++ member), 902
wifi_sta_config_t::listen_interval (C++ member), 901
wifi_sta_config_t::mbo_enabled (C++ member), 902
wifi_sta_config_t::owe_enabled (C++ member), 902
wifi_sta_config_t::password (C++ member), 901
wifi_sta_config_t::pmf_cfg (C++ member), 901
wifi_sta_config_t::reserved (C++ member), 902
wifi_sta_config_t::rm_enabled (C++ member), 902
wifi_sta_config_t::sae_pwe_h2e (C++ member), 902
wifi_sta_config_t::scan_method (C++ member), 901
wifi_sta_config_t::sort_method (C++ member), 901
wifi_sta_config_t::ssid (C++ member), 901
wifi_sta_config_t::threshold (C++ member), 901
wifi_sta_config_t::transition_disable (C++ member), 902
WIFI_STA_DISCONNECTED_PM_ENABLED (C macro), 895
wifi_sta_info_t (C++ struct), 902
wifi_sta_info_t::is_mesh_child (C++ member), 903
wifi_sta_info_t::mac (C++ member), 902
wifi_sta_info_t::phy_11b (C++ member), 902
wifi_sta_info_t::phy_11g (C++ member), 902
wifi_sta_info_t::phy_11n (C++ member), 902
wifi_sta_info_t::phy_lr (C++ member), 903
wifi_sta_info_t::reserved (C++ member), 903
wifi_sta_info_t::rssi (C++ member), 902
wifi_sta_list_t (C++ struct), 903
wifi_sta_list_t::num (C++ member), 903
wifi_sta_list_t::sta (C++ member), 903
WIFI_STATIC_TX_BUFFER_NUM (C macro), 894
WIFI_STATIS_ALL (C macro), 916
WIFI_STATIS_BUFFER (C macro), 916
WIFI_STATIS_DIAG (C macro), 916
WIFI_STATIS_HW (C macro), 916
WIFI_STATIS_PS (C macro), 916
WIFI_STATIS_RXTX (C macro), 916
wifi_storage_t (C++ enum), 923
wifi_storage_t::WIFI_STORAGE_FLASH (C++ enumerator), 923
wifi_storage_t::WIFI_STORAGE_RAM (C++ enumerator), 923
WIFI_TASK_CORE_ID (C macro), 895
WIFI_VENDOR_IE_ELEMENT_ID (C macro), 914
wifi_vendor_ie_id_t (C++ enum), 923
wifi_vendor_ie_id_t::WIFI_VND_IE_ID_0 (C++ enumerator), 923
wifi_vendor_ie_id_t::WIFI_VND_IE_ID_1 (C++ enumerator), 924
wifi_vendor_ie_type_t (C++ enum), 923
wifi_vendor_ie_type_t::WIFI_VND_IE_TYPE_ASSOC_REQ (C++ enumerator), 923
wifi_vendor_ie_type_t::WIFI_VND_IE_TYPE_ASSOC_RES (C++ enumerator), 923
wifi_vendor_ie_type_t::WIFI_VND_IE_TYPE_BEACON (C++ enumerator), 923
wifi_vendor_ie_type_t::WIFI_VND_IE_TYPE_PROBE_REQ (C++ enumerator), 923
wifi_vendor_ie_type_t::WIFI_VND_IE_TYPE_PROBE_RES (C++ enumerator), 923
wl_erase_range (C++ function), 1824
wl_handle_t (C++ type), 1825
WL_INVALID_HANDLE (C macro), 1825
wl_mount (C++ function), 1824
wl_read (C++ function), 1825
wl_sector_size (C++ function), 1825
wl_size (C++ function), 1825
wl_unmount (C++ function), 1824
wl_write (C++ function), 1825
WPS_CONFIG_INIT_DEFAULT (C macro), 937
wps_factory_information_t (C++ struct), 936
wps_factory_information_t::device_name (C++ member), 936
wps_factory_information_t::manufacturer

(C++ *member*), 936
wps_factory_information_t::model_name (C++ *member*), 936
wps_factory_information_t::model_number (C++ *member*), 936
wps_fail_reason_t (C++ *enum*), 930
wps_fail_reason_t::WPS_AP_FAIL_REASON_AUTH (C++ *enumerator*), 930
wps_fail_reason_t::WPS_AP_FAIL_REASON_CONFIG (C++ *enumerator*), 930
wps_fail_reason_t::WPS_AP_FAIL_REASON_MAX (C++ *enumerator*), 930
wps_fail_reason_t::WPS_AP_FAIL_REASON_NORMAL (C++ *enumerator*), 930
WPS_MAX_DEVICE_NAME_LEN (C *macro*), 937
WPS_MAX_MANUFACTURER_LEN (C *macro*), 937
WPS_MAX_MODEL_NAME_LEN (C *macro*), 937
WPS_MAX_MODEL_NUMBER_LEN (C *macro*), 937
wps_type (C++ *enum*), 938
wps_type::WPS_TYPE_DISABLE (C++ *enumerator*), 938
wps_type::WPS_TYPE_MAX (C++ *enumerator*), 938
wps_type::WPS_TYPE_PBC (C++ *enumerator*), 938
wps_type::WPS_TYPE_PIN (C++ *enumerator*), 938
wps_type_t (C++ *type*), 937

X

xEventGroupClearBits (C++ *function*), 1988
xEventGroupClearBitsFromISR (C *macro*), 1992
xEventGroupCreate (C++ *function*), 1986
xEventGroupCreateStatic (C++ *function*), 1986
xEventGroupGetBits (C *macro*), 1994
xEventGroupGetBitsFromISR (C++ *function*), 1992
xEventGroupSetBits (C++ *function*), 1989
xEventGroupSetBitsFromISR (C *macro*), 1993
xEventGroupSync (C++ *function*), 1990
xEventGroupWaitBits (C++ *function*), 1987
xMessageBufferCreate (C *macro*), 2002
xMessageBufferCreateStatic (C *macro*), 2003
xMessageBufferIsEmpty (C *macro*), 2009
xMessageBufferIsFull (C *macro*), 2009
xMessageBufferNextLengthBytes (C *macro*), 2010
xMessageBufferReceive (C *macro*), 2007
xMessageBufferReceiveCompletedFromISR (C *macro*), 2010
xMessageBufferReceiveFromISR (C *macro*), 2008
xMessageBufferReset (C *macro*), 2009
xMessageBufferSend (C *macro*), 2004
xMessageBufferSendCompletedFromISR (C *macro*), 2010
xMessageBufferSendFromISR (C *macro*), 2005
xMessageBufferSpaceAvailable (C *macro*), 2009
xMessageBufferSpacesAvailable (C *macro*), 2010
xQueueAddToSet (C++ *function*), 1943
xQueueCreate (C *macro*), 1944
xQueueCreateSet (C++ *function*), 1942
xQueueCreateStatic (C *macro*), 1945
xQueueGenericCreate (C++ *function*), 1942
xQueueGenericCreateStatic (C++ *function*), 1942
xQueueGenericSend (C++ *function*), 1935
xQueueGenericSendFromISR (C++ *function*), 1939
xQueueGiveFromISR (C++ *function*), 1940
xQueueIsQueueEmptyFromISR (C++ *function*), 1941
xQueueIsQueueFullFromISR (C++ *function*), 1941
xQueueOverwrite (C *macro*), 1949
xQueueOverwriteFromISR (C *macro*), 1951
xQueuePeek (C++ *function*), 1936
xQueuePeekFromISR (C++ *function*), 1937
xQueueReceive (C++ *function*), 1938
xQueueReceiveFromISR (C++ *function*), 1940
xQueueRemoveFromSet (C++ *function*), 1943
xQueueReset (C *macro*), 1953
xQueueSelectFromSet (C++ *function*), 1943
xQueueSelectFromSetFromISR (C++ *function*), 1944
xQueueSend (C *macro*), 1948
xQueueSendFromISR (C *macro*), 1953
xQueueSendToBack (C *macro*), 1947
xQueueSendToBackFromISR (C *macro*), 1951
xQueueSendToFront (C *macro*), 1946
xQueueSendToFrontFromISR (C *macro*), 1950
xRingbufferAddToQueueSetRead (C++ *function*), 2027
xRingbufferCanRead (C++ *function*), 2027
xRingbufferCreate (C++ *function*), 2021
xRingbufferCreateNoSplit (C++ *function*), 2021
xRingbufferCreateStatic (C++ *function*), 2021
xRingbufferGetCurFreeSize (C++ *function*), 2026
xRingbufferGetMaxItemSize (C++ *function*), 2026
xRingbufferPrintInfo (C++ *function*), 2028
xRingbufferReceive (C++ *function*), 2023
xRingbufferReceiveFromISR (C++ *function*), 2023
xRingbufferReceiveSplit (C++ *function*), 2024
xRingbufferReceiveSplitFromISR (C++ *function*), 2024
xRingbufferReceiveUpTo (C++ *function*), 2025

xRingbufferReceiveUpToFromISR (*C++ function*), 2025
xRingbufferRemoveFromQueueSetRead (*C++ function*), 2027
xRingbufferSend (*C++ function*), 2022
xRingbufferSendAcquire (*C++ function*), 2022
xRingbufferSendComplete (*C++ function*), 2023
xRingbufferSendFromISR (*C++ function*), 2022
xSemaphoreCreateBinary (*C macro*), 1954
xSemaphoreCreateBinaryStatic (*C macro*), 1955
xSemaphoreCreateCounting (*C macro*), 1963
xSemaphoreCreateCountingStatic (*C macro*), 1965
xSemaphoreCreateMutex (*C macro*), 1961
xSemaphoreCreateMutexStatic (*C macro*), 1962
xSemaphoreGetMutexHolder (*C macro*), 1967
xSemaphoreGetMutexHolderFromISR (*C macro*), 1967
xSemaphoreGive (*C macro*), 1958
xSemaphoreGiveFromISR (*C macro*), 1960
xSemaphoreGiveRecursive (*C macro*), 1959
xSemaphoreTake (*C macro*), 1956
xSemaphoreTakeFromISR (*C macro*), 1961
xSemaphoreTakeRecursive (*C macro*), 1956
xSTATIC_RINGBUFFER (*C++ struct*), 2028
xStreamBufferBytesAvailable (*C++ function*), 1999
xStreamBufferCreate (*C macro*), 2000
xStreamBufferCreateStatic (*C macro*), 2001
xStreamBufferIsEmpty (*C++ function*), 1999
xStreamBufferIsFull (*C++ function*), 1999
xStreamBufferReceive (*C++ function*), 1996
xStreamBufferReceiveCompletedFromISR (*C++ function*), 2000
xStreamBufferReceiveFromISR (*C++ function*), 1997
xStreamBufferReset (*C++ function*), 1999
xStreamBufferSend (*C++ function*), 1994
xStreamBufferSendCompletedFromISR (*C++ function*), 2000
xStreamBufferSendFromISR (*C++ function*), 1995
xStreamBufferSetTriggerLevel (*C++ function*), 1999
xStreamBufferSpacesAvailable (*C++ function*), 1999
xTaskAbortDelay (*C++ function*), 1910
xTaskCallApplicationTaskHook (*C++ function*), 1919
xTaskCatchUpTicks (*C++ function*), 1931
xTaskCheckForTimeOut (*C++ function*), 1929
xTaskCreate (*C++ function*), 1902
xTaskCreatePinnedToCore (*C++ function*), 1901
xTaskCreateRestricted (*C++ function*), 1905
xTaskCreateStatic (*C++ function*), 1904
xTaskCreateStaticPinnedToCore (*C++ function*), 1903
xTaskDelayUntil (*C++ function*), 1909
xTaskGenericNotify (*C++ function*), 1922
xTaskGenericNotifyFromISR (*C++ function*), 1924
xTaskGenericNotifyStateClear (*C++ function*), 1928
xTaskGenericNotifyWait (*C++ function*), 1925
xTaskGetApplicationTaskTag (*C++ function*), 1918
xTaskGetApplicationTaskTagFromISR (*C++ function*), 1918
xTaskGetHandle (*C++ function*), 1917
xTaskGetIdleTaskHandle (*C++ function*), 1919
xTaskGetTickCount (*C++ function*), 1917
xTaskGetTickCountFromISR (*C++ function*), 1917
xTaskNotify (*C macro*), 1932
xTaskNotifyAndQuery (*C macro*), 1932
xTaskNotifyAndQueryFromISR (*C macro*), 1932
xTaskNotifyAndQueryIndexed (*C macro*), 1932
xTaskNotifyAndQueryIndexedFromISR (*C macro*), 1932
xTaskNotifyFromISR (*C macro*), 1932
xTaskNotifyGive (*C macro*), 1933
xTaskNotifyGiveIndexed (*C macro*), 1933
xTaskNotifyIndexed (*C macro*), 1932
xTaskNotifyIndexedFromISR (*C macro*), 1932
xTaskNotifyStateClear (*C macro*), 1934
xTaskNotifyStateClearIndexed (*C macro*), 1934
xTaskNotifyWait (*C macro*), 1933
xTaskNotifyWaitIndexed (*C macro*), 1933
xTaskResumeAll (*C++ function*), 1916
xTaskResumeFromISR (*C++ function*), 1914
xtensa_perfmon_config (*C++ struct*), 2109
xtensa_perfmon_config::call_function (*C++ member*), 2109
xtensa_perfmon_config::call_params (*C++ member*), 2109
xtensa_perfmon_config::callback (*C++ member*), 2109
xtensa_perfmon_config::callback_params (*C++ member*), 2109
xtensa_perfmon_config::counters_size (*C++ member*), 2110
xtensa_perfmon_config::max_deviation (*C++ member*), 2109
xtensa_perfmon_config::repeat_count (*C++ member*), 2109
xtensa_perfmon_config::select_mask (*C++ member*), 2110
xtensa_perfmon_config::tracelevel (*C++ member*), 2109
xtensa_perfmon_config_t (*C++ type*), 2110
xtensa_perfmon_dump (*C++ function*), 2108

xtensa_perfmon_exec (*C++ function*), 2109
xtensa_perfmon_init (*C++ function*), 2108
xtensa_perfmon_overflow (*C++ function*), 2108
xtensa_perfmon_reset (*C++ function*), 2108
xtensa_perfmon_start (*C++ function*), 2108
xtensa_perfmon_stop (*C++ function*), 2108
xtensa_perfmon_value (*C++ function*), 2108
xtensa_perfmon_view_cb (*C++ function*), 2109
xTimerChangePeriod (*C macro*), 1977
xTimerChangePeriodFromISR (*C macro*), 1983
xTimerCreate (*C++ function*), 1967
xTimerCreateStatic (*C++ function*), 1969
xTimerDelete (*C macro*), 1978
xTimerGetExpiryTime (*C++ function*), 1975
xTimerGetPeriod (*C++ function*), 1975
xTimerGetTimerDaemonTaskHandle (*C++ function*), 1972
xTimerIsTimerActive (*C++ function*), 1972
xTimerPendFunctionCall (*C++ function*), 1974
xTimerPendFunctionCallFromISR (*C++ function*), 1973
xTimerReset (*C macro*), 1979
xTimerResetFromISR (*C macro*), 1984
xTimerStart (*C macro*), 1976
xTimerStartFromISR (*C macro*), 1981
xTimerStop (*C macro*), 1976
xTimerStopFromISR (*C macro*), 1982